Specifying and Verifying Hardware-based Security Enforcement Mechanisms

Thomas Letan

UNIVERSITE BRETAGNE LOIRE / MATHSTIC

CentraleSupélec

THESE DE DOCTORAT DE

CENTRALESUPELEC RENNES
COMUE UNIVERSITE BRETAGNE LOIRE

ECOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Thomas Letan

Specifying and Verifying Hardware-based Security Enforcement Mechanisms

Thèse présentée et soutenue à Paris, le 25 octobre 2018
Unité de recherche : CIDRE
Thèse N° : 2018-06-TH

Rapporteurs avant soutenance :

Gilles Barthe	Professor, IMDEA Software Institute
Laurence Pierre	Professeur, Université Grenoble Alpes

Composition du Jury :

Présidente

Emmanuelle Encrenaz	Maître de conférences, Sorbonne Université

Membres

Gilles Barthe	Professor, IMDEA Software Institute
Laurence Pierre	Professeur, Université Grenoble Alpes
Pierre Chifflier	Chef de laboratoire, ANSSI
Guilaume Hiet	Maître de conférences, CentraleSupélec Rennes

Directeur de thèse

Ludovic Mé	Professeur, Inria Rennes

Invité

Alastair Reid	Researcher, ARM Ltd.

Abstract

In this thesis, we consider a class of security enforcement mechanisms we called *Hardware-based Security Enforcement* (HSE). In such mechanisms, some trusted software components rely on the underlying hardware architecture to constrain the execution of untrusted software components with respect to targeted security policies. For instance, an operating system which configures page tables to isolate userland applications implements a HSE mechanism.

For a HSE mechanism to correctly enforce a targeted security policy, it requires both hardware and trusted software components to play their parts. During the past decades, several vulnerability disclosures have defeated HSE mechanisms. We focus on the vulnerabilities that are the result of errors at the specification level, rather than implementation errors. In some critical vulnerabilities, the attacker makes a legitimate use of one hardware component to circumvent the HSE mechanism provided by another one. For instance, cache poisoning attacks leverage inconsistencies between cache and DRAM's access control mechanisms. We call this class of attacks, where an attacker leverages inconsistencies in hardware specifications, *compositional attacks*.

Our goal is to explore approaches to specify and verify HSE mechanisms using formal methods that would benefit both hardware designers and software developers. Firstly, a formal specification of HSE mechanisms can be leveraged as a foundation for a systematic approach to verify hardware specifications, in the hope of uncovering potential compositional attacks ahead of time. Secondly, it provides unambiguous specifications to software developers, in the form of a list of requirements.

Our contribution is two-fold:

- We propose a theory of HSE mechanisms against hardware architecture models. This theory can be used to specify and verify such mechanisms. To evaluate our approach, we propose a minimal model for a single core x86-based computing platform. We use it to specify and verify the HSE mechanism provided by Intel to isolate the code executed while the CPU is in System Management Mode (SMM), a highly privileged execution mode of x86 microprocessors. We have written machine-checked proofs in the Coq proof assistant to that end.

- We propose a novel approach inspired by algebraic effects to enable modular verification of complex systems made of interconnected components as a first step towards addressing the challenge posed by the scale of the x86 hardware architecture. This approach is not specific to hardware models, and could also be leveraged to reason about composition of software components as well. In addition, we have implemented our approach in the Coq theorem prover, and the resulting framework takes advantages of Coq proof automation features to provide general-purpose facilities to reason about components interactions.

Keywords: Security • Hardware Verification • Formal Specification • Formal Methods • Coq

Remerciements

Je tiens à remercier en premier lieu les membres de mon jury de thèse, à commencer par Gilles Barthe et Laurence Pierre pour avoir rapporté cette thèse. Merci aussi à Emmanuelle Encrenaz-Tiphene et Alastair Reid. Je suis honoré de l'intérêt que vous avez porté à mes travaux.

Je remercie ensuite Ludovic Mé, mon directeur de thèse. Venir te voir à la fin de l'un de tes cours pour te demander s'il n'était pas possible de faire un stage « sur le kernel » aura été avec le recul mon premier vrai pas en direction du monde de la recherche. Merci de m'avoir proposé de rester un an de plus dans l'équipe CIDRE après la fin de mon stage et merci, surtout, de m'avoir parlé de l'ANSSI et encouragé à y postuler. Pierre, Guillaume, votre encadrement et votre soutien tout au long de cette thèse expliquent pour beaucoup sa qualité finale, quand bien même la direction de mes recherches s'est finalement révélée assez éloignée de vos domaines de prédilection. Merci de m'avoir laissé libre d'explorer les sujets qui m'intéressaient. Je n'oublierai jamais comment vous avez su à plusieurs reprises m'expliquer mon propre travail, avec ces mots que je n'arrivais pas à trouver. Merci, donc, à Guillaume Hiet. Début 2013, tu acceptais avec Frédéric Tronel de m'encadrer une première fois pour mon stage de fin d'études. J'étais loin de me douter, alors, que nous travaillerions ensemble aussi longtemps, ni combien notre collaboration m'apporterait. Merci, ensuite, à Pierre Chifflier. Nos discussions pendant ces quatre années ont nourri ma curiosité et forgé mon esprit critique. Te voir t'intéresser à tant de sujets a été une véritable source d'inspiration. Merci, enfin, Benjamin Morin, pour m'avoir fait confiance. L'évolution de ta carrière ne nous a pas permis de travailler ensemble jusqu'au bout de cette thèse, mais je n'oublie pas que c'est en grande partie grâce à toi qu'elle a pu débuter.

Tout au long de cette thèse, j'ai pu compter sur le soutien de beaucoup de personnes. Je tiens ainsi à remercier chaleureusement Yves-Alexis Perez. En ta qualité de chef d'équipe, tu as dû composer avec un agent dont les disponibilités n'étaient pas toujours dépendantes des tâches que tu lui confiais. Si j'ai pu tout à la fois conduire avec succès un travail de thèse et participer aux missions de l'Agence, c'est aussi grâce à toi. J'ai souvent été sensible au temps de recherche que tu me ménageais, à la liberté que tu me laissais pour choisir mes sujets et à la confiance que tu m'as témoignée en me confiant des missions passionnantes. Merci à Arnaud Fontaine. Je n'ai pas oublié notre conversation au Dernier Métro, pendant la rédaction de mon premier article, où nous avons échangé sur mes travaux. Tu as su répondre à mes questions et mettre des mots — qui se retrouvent en filigrane dans tout ce manuscrit — sur mes idées confuses d'alors. Merci à Pierre Néron. Tu m'as énormément aidé à revoir ma copie après le refus de ma première soumission ; tes conseils et relectures ne sont pas étrangers à l'acceptation de mon premier article. Merci aussi à Yann Régis-Gianas d'avoir accepté de me rencontrer pour que je lui présente mes travaux. Ton aide aura été précieuse pour avoir la chance de pouvoir présenter mes travaux à Oxford.

Merci à Anaël Beaugnon. Ton soutien, ton aide, tes encouragements m'ont indéniablement aidé à tenir ce marathon qu'est une thèse. Tu t'es toujours spontanément proposée pour relire mes articles et ce manuscrit, quand bien même ils n'étaient pas tout à fait aligné avec ton domaine de recherche. Ton courage et ta ténacité face à l'adversité auront quant à eux été une véritable source d'inspiration. Merci à Marion Daubignard, pour ta gentillesse, ta bonne humeur et tes conseils. J'ai précieusement gardé la citation de ton arrière grand-mère : elle trône encore sur le côté de mon écran. Je la regarde souvent et elle m'aide à me remotiver quand j'en ai besoin. Je ne peux pas ne pas remercier Aurélien Deharbe, qui appréciera je l'espère cette double négation à sa juste valeur. Je suis heureux que nos pérégrinations respectives nous aient amenés à nous rencontrer. J'ai toujours pu compter sur ton soutien, ton énergie et ta bonne humeur communicative quand j'en avais besoin.

Je remercie mes collègues à l'ANSSI. Depuis mon arrivée en octobre 2014, j'ai eu la chance de travailler avec des personnes extrêmement compétentes et passionnées par ce qu'elles faisaient. Vous voir vous investir dans vos missions a nourri ma propre motivation à mener les miennes du mieux possible. J'ai une pensée toute particulière pour ceux avec qui j'ai partagé un bureau. J'en ai déjà cité plusieurs déjà, mais il me faut nommer Alain Ozanne et Philippe Thierry pour être complet. Je remercie aussi chaleureusement l'équipe CIDRE, pour qui je garderai pendant longtemps encore une affection particulière. C'est en vous côtoyant que j'ai découvert le monde de la recherche. La bienveillance et l'investissement dont vous avez tous fait sans cesse preuve forcent le respect. J'ai toujours pris beaucoup de plaisir à vous retrouver et vous présenter l'avancée de mes travaux pendant nos séminaires d'équipe et je regrette seulement ne pas avoir eu l'occasion de le faire plus souvent. J'ai préféré ne pas citer vos noms à tous, car vous êtes trop nombreux. Je me connais et je sais que le risque est grand que j'oublie l'un d'entre vous.

Je ne peux pas conclure ces remerciements sans prendre le temps d'exprimer ma gratitude à ma famille, qui a toujours su répondre présente ; aussi bien dans les petits riens du quotidien que dans les étapes clefs de ma vie. Je crois que je ne mesurerai jamais vraiment tout ce que vous avez fait pour moi. Maman, je crois que je ne t'ai jamais explicitement remercié d'avoir pris la décision de mettre entre parenthèses ta carrière professionnelle pendant si longtemps pour tes enfants. Il m'a fallu longtemps pour prendre la mesure de tout ce que cela représentait. Je profite donc de l'opportunité que m'offre ce manuscrit pour me rattraper. Papa, je t'ai toujours vu redoubler d'efforts pour pourvoir aux besoins des tiens. Tu voulais, toi aussi, t'assurer que tes enfants puissent faire leurs choix de vie, sans regret. Il est ici utile de le dire : vous avez, l'un comme l'autre et l'un avec l'autre, réussit. C'est à nous, maintenant, de tracer le reste de notre route sans jamais oublier la bonne fortune qui fut la nôtre d'être si bien lotis à notre point de départ. N'est-ce pas, Marion, ma petite sœur que j'aime tant ? Merci à toi aussi, évidemment. Je sais que, peu importe ce qui m'arrive, peu importe mes choix de vie, tu seras là.

Enfin, je me tourne vers toi, Juliette, et les mots me manquent pour exprimer toute ma reconnaissance. Pendant la rédaction de ce manuscrit, mais surtout pendant une étape charnière et difficile de ton parcours personnel, tu as toujours cherché à me soutenir, à m'encourager, à me rassurer. J'ai pu me reposer sur ta confiance quand la mienne vacillait. C'est donc tout naturellement que je te réserve mes ultimes remerciements et te dédie ce manuscrit.

Contents

Notations

Constructors We often define sets of values in terms of functions to construct these values. These functions are called "constructors," and they have mutually exclusive images, i.e. it is not possible to construct the same value with two different constructors. Constructor names begin with a capital letter. We adopt a notation similar to Haskell sum types to define sets *via* constructors. For instance, this is how we define the disjoint union operator $\uplus$:

$$\begin{aligned} A \uplus B \quad &\triangleq \quad \mathit{Left} : A \to A \uplus B \\ &\mid \quad \mathit{Right} : B \to A \uplus B \end{aligned}$$

Given $a \in A$, we write $\mathit{Right}(a) \in A \uplus B$ for the injection of a inside $A \uplus B$.

Named Tuples We adopt a notation similar to Haskell record types to manipulate "named" tuples, that is tuples where each component is a field identified by a name.

$$T \triangleq \langle\, \mathit{field}_1 : A, \quad \mathit{field}_2 : B \,\rangle$$

For $x \in T$, we write $x.\mathit{field}_1$ for selecting the value of the field_1 field. We write $x\{\mathit{field}_i \leftarrow a\}$ for updating the value of the field field_i. As a consequence,

$$x\{\mathit{field}_i \leftarrow a\}.\mathit{field}_k = \begin{cases} a & \text{if } k = i \\ x.\mathit{field}_k & \text{otherwise} \end{cases}$$

We use a similar notation for functions, that is given $f : A \to B$, $f\{x \leftarrow y\}$ is a new function such that

$$\forall z \in A, f\{x \leftarrow y\}(z) = \begin{cases} y & \text{if } z = x \\ f(z) & \text{otherwise} \end{cases}$$

Finally, we ease the definition of nested updates by gathering nested fields on the left side of $\leftarrow$. Let $R \triangleq \langle s : T \to U, \quad v : W \rangle$ be a set of named tuples whose field s is a function which maps elements of T to element of U. To replace the value associated to $t \in T$ by $u \in U$ in the function associated with the field s of $r \in R$, we write $r\{s(t) \leftarrow u\}$ instead of $r\{s \leftarrow r.s\{t \leftarrow u\}\}$.

Résumé de la thèse

Dans le cadre de cette thèse, nous nous intéressons à une classe particulière de mécanismes de sécurité. Ces mécanismes nécessitent qu'un ou plusieurs logiciels de confiance configurent la plate-forme matérielle afin de contraindre l'exécution du reste de la pile logicielle à respecter une politique de sécurité donnée. Nous qualifierons par la suite de mécanisme HSE (de l'anglais *Hardware-based Security Enforcement*) les instances de cette classe. L'utilisation par le système d'exploitation des mécanismes de pagination pour isoler chaque application est sûrement l'exemple le plus courant de la mise en pratique d'un mécanisme HSE.

Contexte

Le contournement d'une politique de sécurité normalement assurée par le biais d'un mécanisme HSE peut s'expliquer par une erreur dans sa mise en œuvre, aussi bien dans l'un des logiciels de confiance chargés de sa configuration que dans les mécanismes matériels sur lesquels il se repose. Nous nous sommes intéressés à ce second cas, et plus spécifiquement encore au problème des erreurs affectant les spécifications de la plate-forme. Cette dernière est composée de plusieurs dizaines de composants interagissant ensemble et la complexité résultante de ces interactions rend en pratique difficile la conception d'un mécanisme à visée sécuritaire. Pour chaque nouvelle fonctionnalité que le concepteur de la plate-forme désire ajouter, il est nécessaire de s'assurer (1) qu'elle n'interfère pas avec les mécanismes HSE existants et (2) qu'il n'est pas possible de la contourner par le biais d'une fonctionnalité antérieure. Ces risques peuvent être critiques pour la sécurité de la plate-forme comme l'illustre l'attaque `SENTER` *Sandman* présentée en 2015 par Xeno Kovah *et al.* [1]. Les auteurs ont montré qu'il était possible de modifier le contenu de la mémoire flash d'un ordinateur en utilisant une extension de l'architecture x86 nommée Intel TXT [2]. Ils ont en effet remarqué que l'instruction `SENTER` permettait — dans sa première version — de désactiver une composante essentielle du mécanisme de protection en intégrité de la mémoire flash.

La multiplication des chemins d'attaque potentiels liée au nombre grandissant des composants constitue une menace clairement identifiée dans la littérature [3]. Dans le cas particulier des mécanismes HSE, les vulnérabilités successives affectant l'architecture x86 [4, 5, 6, 7, 1] ont été caractérisées par une sévérité très importante, car ces mécanismes visent à assurer l'isolation des couches les plus basses — et, par voie de conséquence, les plus privilégiées — de la pile logicielle.

Objectifs

Dans cette thèse, nous avons cherché à proposer une approche rigoureuse pour spécifier et vérifier, par le biais de méthodes formelles, des mécanismes HSE. Notre hypothèse de départ est qu'une telle approche bénéficierait à la fois aux concepteurs des plates-formes matérielles et aux développeurs de logiciels qui s'appuient sur les mécanismes matériels de ces plates-formes. Les premiers pourraient vérifier que leurs mécanismes matériels permettent effectivement de mettre en œuvre les politiques de sécurité visées. Quant aux seconds, ils pourraient profiter de spécifications décrivant sans ambiguïtés les exigences auxquelles leurs logiciels doivent se conformer pour pouvoir profiter de ces politiques.

Notre démarche se place à la croisée de deux domaines de vérification. La vérification matérielle, d'une part, se concentre généralement sur des propriétés qui s'appliquent inconditionnellement à la plate-forme comme à la pile logicielle exécutée par cette dernière. De nombreux travaux ont ainsi cherché à vérifier des protocoles de cohérence de caches [8, 9], ou la correction de l'implémentation d'un modèle mémoire par un processeur [10]. La vérification de logiciels bas niveaux, notamment des systèmes d'exploitation ou des hyperviseurs, se repose naturellement sur des modèles de la plate-forme matérielle sous-jacente. Néanmoins, ces modèles abstraient bien souvent autant que faire se peut la complexité de l'architecture matérielle, pour n'en garder que les éléments essentiels — bien souvent, les mécanismes de pagination et les interruptions. La conséquence de cet état de fait est que les mécanismes nécessitant une configuration logicielle sont moins souvent les sujets de vérification formelle.

Les travaux qui se rapprochent le plus de notre objectif et dont nous avons connaissance sont ceux de Jomaa *et al.* [11], en lien avec le protokernel Pip [12]. Nous nous inscrivons dans la continuité de cette approche, mais cherchons à dégager un formalisme beaucoup plus générique, qui reposerait notamment sur un modèle matériel le plus générique possible. Cependant, un tel modèle n'est pas sans poser de sérieux défis quant à son applicabilité dans un problème de vérification réaliste. En effet, la complexité d'un modèle a un impact direct sur la facilité avec laquelle on peut l'exploiter. Il est donc important de se poser, en amont, les bonnes questions quant à l'approche utilisée pour le définir, afin que les efforts nécessaires pour sa conception ne soient pas dépensés en vain. Plusieurs travaux ont plaidé en faveur d'une approche basée sur un raisonnement par composition (*compositional reasoning* en anglais), où le système est divisé en un sous-ensemble de composants et la vérification axée autour de leurs interactions, pour faire face à ces défis [13, 14].

Contributions

Dans cette thèse, nous présentons deux contributions complémentaires, qui ont chacune fait l'objet d'une publication à la conférence *Formal Methods* ; d'abord en 2016 [15], puis en 2018 [16].

Une théorie des mécanismes HSE. Notre première contribution est une théorie des mécanismes HSE, dont l'objectif premier est de servir de support à la spécification et à la vérification de ces derniers. Elle s'articule autour d'une méthodologie divisée en plusieurs étapes. L'architec-

ture matérielle est dans un premier temps modélisée sous la forme d'un système de transitions étiquetées (*labeled transition system*, en anglais, désigné par la suite par l'acronyme LTS). Un LTS est traditionnellement caractérisé par un ensemble d'états, un ensemble d'étiquettes et une relation de transition. Une étiquette est attachée à chaque transition pour permettre de leur donner une sémantique particulière. Le plus souvent, l'étiquette permet de décrire ce qui a causé la transition.

Une trace d'un LTS est une séquence potentiellement infinie de transitions et décrit un comportement possible du système. Dans le cas qui nous intéresse, une trace décrit une exécution d'une pile logicielle par la plate-forme. La formalisation de politiques de sécurité pour des systèmes de transitions est désormais bien établie. Ces politiques peuvent être formellement définies sous la forme de prédicats de traces [17, 18, 18, 19] ou, dans les cas les plus complexes, d'ensemble de traces [20].

Un modèle matériel se présente sous la forme d'un quadruplet $\langle H, L_S, L_H, \rightarrow \rangle$ dans notre théorie, où :

- H est l'ensemble des états que peut prendre le LTS, par exemple la valeur des registres des différents composants matériels de la plate-forme et le contenu de la DRAM ;
- L_S est l'ensemble des étiquettes attachées aux transitions dites logicielles, qui sont une conséquence directe et prévisible de l'exécution, par la plate-forme matérielle, d'une instruction faisant partie du programme d'un logiciel ;
- L_H est l'ensemble des étiquettes attachées aux transitions dites matérielles ;
- $\rightarrow$ est la relation de transition du système.

En nous basant sur ce type de modèle matériel, nous pouvons spécifier un mécanisme HSE sous la forme d'un ensemble de logiciels de confiance chargés d'implémenter ce mécanisme et d'exigences qu'ils doivent respecter pendant leurs exécutions. À partir de cette définition, il est possible de dégager un sous-ensemble de traces du modèle matériel dans lesquelles les logiciels de confiance ont correctement implémenté le mécanisme HSE étudié, en respectant à tout moment les deux exigences. Par la suite, un mécanisme HSE peut être prouvé correct vis-à-vis de la politique de sécurité qu'il cherche à mettre en œuvre, si le sous-ensemble des traces du modèle matériel qui le caractérisent satisfait le prédicat formalisant la politique.

Nous avons déroulé notre méthodologie sur un exemple réel, à savoir le mécanisme Hardware-based Security Enforcement (HSE) implémenté par le BIOS des plates-formes x86 pour isoler leur exécution de celles des logiciels appartenant au reste de la pile logicielle. Le BIOS (*Basic Input/Output System*) est un composant logiciel fourni par le constructeur de la plate-forme, dont le principal objectif est d'initialiser puis de maintenir cette dernière en état de fonctionnement. Ce mécanisme HSE repose sur des fonctionnalités matérielles relativement peu connues, notamment un contexte d'exécution particulier des processeurs d'Intel — au même titre que les *rings*, par exemple — nommé le *System Management Mode* (SMM). Le SMM a ceci d'intéressant qu'il est le contexte d'exécution le plus privilégié de l'architecture x86, si bien qu'une escalade de privilège permettant d'exécuter du code malveillant en SMM peut

avoir des conséquences désastreuses pour la sécurité de la plate-forme. Malheureusement, il a été l'objet en dix ans de plusieurs vulnérabilités profitant d'incohérences dans les différents composants matériels impliqués dans son isolation [4, 5, 6]. Parce que le SMM et les autres mécanismes matériels impliqués dans le mécanisme HSE qui nous intéresse ne sont pas pris en compte dans les modèles x86 de notre connaissance, nous avons dû en définir un nouveau. Cette démarche s'est révélée riche d'enseignement quant aux propriétés qu'un modèle générique d'une plate-forme matérielle devrait exhiber pour que ce dernier demeure utilisable dans un processus de vérification. La modularité, tant de la modélisation que du travail de vérification est, de notre point de vue, l'élément clef à privilégier pour pouvoir accompagner au mieux un passage à l'échelle.

Raisonnement par composition pour Coq. Le raisonnement par composition permet de vérifier chaque composant d'un système complexe en isolation, en spécifiant des hypothèses sur le comportement du reste du système d'une part et en s'assurant que le composant adoptera bien un comportement attendu en retour. Une fois cette première étape réalisée pour chaque composant, il devient possible de raisonner sur la composition de chacun des éléments : si le comportement garanti pour un composant C_1 satisfait les hypothèses de raisonnement d'un second composant C_2, alors il est possible de conclure que la composition de C_1 et de C_2 garantit le comportement de C_2.

Notre seconde contribution est une approche permettant la conduite de raisonnement par composition sur des composants modélisés grâce à un langage purement fonctionnel [16] ainsi qu'une implémentation nommée FreeSpec de cette approche [21] dans l'assistant de preuve Coq [22]. L'originalité de notre contribution est de mettre à profit des paradigmes fonctionnels — les monades [23], les effets algébriques et les *handlers* d'effets [24] — pour les appliquer au domaine de la vérification de plates-formes matérielles. Les modèles écrits dans notre formalisme sont facilement lisibles et soulignent bien les connexions entre les différents composants. De plus, nous avons implémenté des tactiques dédiées permettant d'automatiser en partie l'exploration de ces modèles, ce qui facilite grandement le travail de vérification.

Rien ne cantonne cependant le résultat de ce travail à la vérification de mécanismes HSE. Il peut ainsi tout à fait s'appliquer dans le cadre de la vérification d'un système purement logiciel, dès lors que ce logiciel peut être décrit sous la forme de plusieurs composants interagissant ensemble. La principale limitation de notre approche tient dans les contraintes que nous imposons aux interconnexions des composants du système. En l'état, FreeSpec ne permet par exemple pas de considérer des graphes qui contiennent des cycles ou des « arêtes en avant » (*forward edges* en anglais). Cette contrainte est importante, mais dans la pratique, nous avons constaté qu'il est possible d'étudier une architecture matérielle à un niveau de détail où les composants s'organisent en arbre, selon une hiérarchie par couches successives.

Travaux futurs

La suite logique de nos travaux serait d'appliquer le formalisme de FreeSpec dans le cadre de

la vérification complète d'un mécanisme HSE par le biais de notre théorie. Par exemple, nous pourrions remplacer le modèle matériel que nous avons développé pour spécifier et vérifier le mécanisme HSE implémenté par le BIOS et juger l'impact que cela aurait sur le travail de vérification. Nous ne doutons par ailleurs pas que les limitations actuelles de l'approche que nous proposons avec FreeSpec se heurteront à la complexité inhérente aux architectures matérielles. Les résoudre nous permettrait d'envisager le passage à l'échelle et donc la définition d'un modèle x86 générique, pouvant servir de support à une large collection de mécanismes HSE couramment implémentés par les couches basses de la pile logicielle. Cela poserait très vite la question de la confiance que l'on peut placer dans un tel modèle et notamment son adéquation avec ce qui est réellement implémenté par le matériel. La validation de modèle est un problème de recherche à part entière. Dans notre cas, elle est rendue plus complexe par le fait que beaucoup de composants matériels sont fortement intégrés à la plate-forme matérielle et difficilement accessible séparément. C'est par exemple le cas du *Platform Controller Hub* (PCH), le composant matériel qui fait l'intermédiaire entre le processeur et les périphériques les moins demandant en vitesse d'accès. Ce dernier est maintenant directement intégré directement dans la puce des processeurs Intel.

1

Introduction

"All problems in computer science can be solved by another level of indirection."

— David Wheeler

To manage complexity, computing platforms are commonly built as successions of abstraction layers, from the hardware components to high-level software applications. Each layer leverages the interface of its predecessor to expose a higher-level, more constrained set of functionalities for its successors. This enables separation of concerns —each layer encapsulates one dimension of the overall complexity— and modularity —two layers which expose the same interface can be seamlessly interchanged.

From a security perspective, each layer is often more privileged than its successors. For instance, system software components (*e.g.* an Operating System (OS) or a hypervisor) manage the life cycle of upper layers (*e.g.* applications or guest OSes). In such a context, one layer implicitly trusts its predecessors. On the one hand, it is important to keep this fact in mind when we consider the security of the computing platform. Trust in lower levels of abstraction can be misplaced, *e.g.* hardware implants and backdoors pose a significant threat to the platform security [25], and the platform firmware has been used as a persistent attack vector across machine restarts [26]. On the other hand, one layer may constrain the execution of its successors, with respect to a targeted security policy, *e.g.* an OS enforces availability —fair share of processor time—, confidentiality and integrity —exclusive partition of physical memories— properties for the applications it manages. Concerning the lowest layers of software stack, *e.g.* the Basic Input/Output System (BIOS) and system software components, the common approach is to rely on features provided by the hardware architecture to reduce the hardware capabilities that can be used by upper layers. This scenario characterizes a class of security mechanisms we call HSE mechanisms.

In the following, we first detail in more detail how HSE mechanisms are implemented and which threats these implementations face. In this thesis, we aim to specify and verify HSE

mechanisms. We motivate this choice, then give a brief overview of our contributions. We conclude this first Chapter with a brief summary of the outline of this manuscript.

1.1 Hardware-based Security Enforcement Mechanisms

A HSE mechanism consists of the configuration by a trusted software component of the underlying hardware architecture in order to constrain the execution of untrusted software components with respect to a targeted security policy. For instance, system software components often leverage, among other mechanisms, a Memory Management Unit (MMU) to partition the system memory, and therefore needs to setup so-called page tables. Thus, when an application is executed, it can only access a subset of the system memory. Besides, the processor can leverage a hardware timer to stop applications execution, without the need for these applications to cooperate.

A HSE mechanism enforces its targeted security property when (1) the trusted software components correctly configure the hardware features at their disposal, and (2) these features are sufficient to constrain the untrusted software execution as expected. Both requirements remain challenging and have been violated at many occasions in the past, due to software and hardware errors alike.

Software Errors. Part of vulnerabilities in HSE mechanism implementations by trusted software components are due to some misuse of hardware features [27]. In the past, most lower-level pieces of software, such as firmware components, have not been conceived and implemented with security as primary focus. The increasing complexity of hardware architectures can also be held partly responsible. While computers are made of dozens of components, software developers have to read and understand as many, independent and often large documents of various forms (*e.g.* data sheets, developer manuals), and they rarely focus on security. When software developers misunderstand the documentation, as it happened for instance for the `mov ss` and `pop ss` x86 instructions [28], the impact in terms of security can be significant.

Hardware Errors. Over the past decades, vendors have regularly added security features to their products. Intel, for instance, has notably introduced hardware-based virtualization (VT-x, VT-d) [29], dynamic root of trust (TXT) [2], or applicative enclaves (SGX) [30, 31]. It is crucial to notice that most of them have been circumvented due to implementation bugs [32, 33]. This is not surprising, as novel hardware features tend to be more and more complex. In extreme scenarios, novel hardware features have been turned into attack vectors [32]. In addition to these implementation errors, the fact that hardware architectures often comprise hundreds of features implemented by dozens of interconnected devices complicates the conception of new hardware features. Indeed, these new features should not interfere with the security properties enforced by the existing components of the platform. For instance, the flash memory (where lives the BIOS code) is supposedly protected against arbitrary write accesses from system software components, thanks to a particular hardware interrupt. When Intel introduced

TXT [2], they did not anticipate that this novel security feature had the particular side effect of disabling the hardware interrupt used to protect the flash memory [1]. Such inconsistencies in hardware specifications pave the road toward compositional attacks [3]. Compositional attacks characterize scenarios where each component is working as expected in isolation, yet their composition creates an attack path which prevents end-to-end security enforcement. In the context of HSE mechanisms, this means untrusted software components can leverage one hardware component to defeat a HSE mechanism implemented to constrain its execution. Compositional attacks are due to a flaw in the specifications of the computing platform. As such, they precede implementation errors, and their countermeasures often require a change in the hardware interface. To prevent them, it is mandatory to reason about the computing platform as a whole.

1.2 Formal Verification of HSE Mechanisms

The significant impact of previously disclosed compositional attacks [4, 5, 6, 7, 1] motivate our desire to formally specify and verify HSE mechanisms. We believe this would benefit both hardware designers and software developers. Firstly, a formal specification of HSE mechanisms can be leveraged as a foundation for a systematic approach to verify hardware specifications. For each novel hardware feature introduced, it is necessary to check that the previous proofs hold, meaning this feature does not introduce any compositional attack. Secondly, it provides unambiguous specifications to firmware and system software developers, in the form of a list of requirements to comply with, and the provided security properties. We believe these specifications are a valuable addition to the existing documentation, because they gather at one place information that is normally scattered across many documents which sometimes suffer from lack of security focus.

We steer a middle course between two domains: hardware and system software verification. Generally, hardware verification focuses on properties that are transparent to the executed software, and require no configuration from its part, while software verification abstracts as much as possible the hardware architecture complexity. However, compositional attacks can come from unsuspected places, with no apparent link with the considered security mechanism. For instance, the `SENTER` Sandman attack [1] leveraged a dedicated execution mode of x86 processor to disable the protection of the flash memory wherein the BIOS code is stored. Hence, the composition of the numerous configurable hardware features is less subject to formal verification. At the same time, some components are individually complex: for example, the Intel Architectures Software Developer Manual [34] is 4842 pages long, the Memory Controller Hub datasheet [35] is 430 pages long, and the Platform Controller Hub datasheet [36] is 988 pages long. Besides, new hardware components and new versions of already existing components are frequently released. As a consequence, the more modular our models and proofs are, the more practicable our approach becomes. Otherwise, each modification of the hardware architecture will have an important impact on the proofs already written.

1.3 Contributions

During the first stage of this thesis, we propose a theory of HSE mechanisms in the form of requirements trusted software components have to satisfy. A HSE mechanism is correct if its requirements are sufficient to make the hardware architecture enforce a targeted security policy. To evaluate our approach, we formally specify and verify a HSE mechanism implemented by the BIOS of x86-based computing platform at runtime to remain isolated from the rest of the software stack. Our choice has been motivated by the prominent position of the Intel hardware architecture on the personal computer market. Moreover compositional attacks targeting this architecture have been disclosed [4, 5, 6, 7, 1]. The resulting model assumes as little as possible about the actual implementation of the BIOS, and constitutes, to the extent of our knowledge, the first formalization of the BIOS security model at runtime. Our model and its related proofs of correctness have been implemented in the Coq theorem prover, to increase our confidence in our result. This work has been presented at the 21^{th} International Symposium on Formal Methods (FM2016) [15]. In addition, the resulting project, called SpecCert, has been made available as free software [37].

After this first contribution, we focus our attention on the challenge posed by the modeling of a complex hardware architecture like the x86 architecture. In this thesis, we advocate for the use of a general-purpose model of a hardware architecture to support the specification of HSE mechanisms which can be implemented on this architecture. Our experiment with our theory of HSE mechanisms convinced us that such a general-purpose model should obey certain requirements, notably in terms of readability and modularity, so it could remain applicable to real-life verification problem. Our second contribution is a novel approach, inspired by algebraic effects, to enable modular verification of complex systems made of interconnected components. This approach is not specific to hardware models, and could also be leveraged to reason about composition of software components as well. This work has been presented at the 22^{th} International Symposium on Formal Methods (FM2018) [16]. Besides, we have implemented our approach in Coq, and the resulting framework, called FreeSpec, takes advantages of theorem prover automation features to provide general-purpose facilities to reason about components interactions. Similarly to SpecCert, FreeSpec has been published as free software [21].

1.4 Outline

The rest of this manuscript proceeds as follows.

First of all, Part I provides the context in which this thesis falls. In Chapter 2, we give an introduction to the x86 hardware architecture and the particular role played by the BIOS, in order to present three illustrative compositional attacks. In Chapter 3, we review existing formal verification approaches that have been proposed to verify hardware and software systems. We justify, in this context, our choices in terms of formalism and tools.

Part II focuses on our first contribution. In Chapter 4, we present our general-purpose theory to support the specification and verification of HSE mechanisms. In Chapter 5, we leverage this

theory to reason about the HSE mechanism implemented by the BIOS to remain isolated from the rest of the software stack at runtime.

Part III focuses on our second contribution. In Chapter 6, we present our compositional reasoning framework for Coq, to modularly specify and verify systems made of interconnected components

Finally, we conclude this thesis in Chapter 7, where we suggest some possible directions for future work.

Part I

Context

2

Intel x86 Architecture and BIOS Background

"You're building your own maze, in a way, and you might just get lost in it."

— Marijn Haverbeke

From a market share perspective, the x86 hardware architecture is widely used for laptops, desktops and servers. Intel has introduced several security features to support HSE mechanism over the past decades: hardware-based virtualization (VT-x, VT-d) [34, Volume 3, Chapter 23], dynamic root of trust (TXT) [2], or applicative enclaves (SGX) [34, Volume 3, Chapter 36][31]. As for the BIOS, it is the most privileged piece of software executed by the hardware architecture, and implements several HSE mechanisms to remain isolated from the rest of the software stack. The correctness of these mechanisms is therefore of key importance; yet they have been the object of several compositional attacks [4, 5, 6, 7, 1]. These attacks have motivated our effort to provide a formal framework for reasoning about HSE mechanisms. However, it is important to emphasize that other mainstream architectures (*e.g.* ARM) work in a similar basis and potentially suffer similar issues. Our contributions are thus intended to be applicable to other hardware architectures.

The rest of this Chapter proceeds as follows. We provide the necessary details on how a typical x86 hardware architecture works (Section 2.1) and we explain why and how the BIOS remains isolated from the rest of the software stack at runtime (Section 2.2). This background allows us to describe the critical compositional attacks we already mentioned (Section 2.3).

2.1 Introduction to x86 Architecture

Describing hardware architectures in depth is challenging, because they tend to comprise an increasing number of interconnected components of various natures. The x86 hardware architecture is a perfect illustration of this, which is probably best demonstrated by the scale

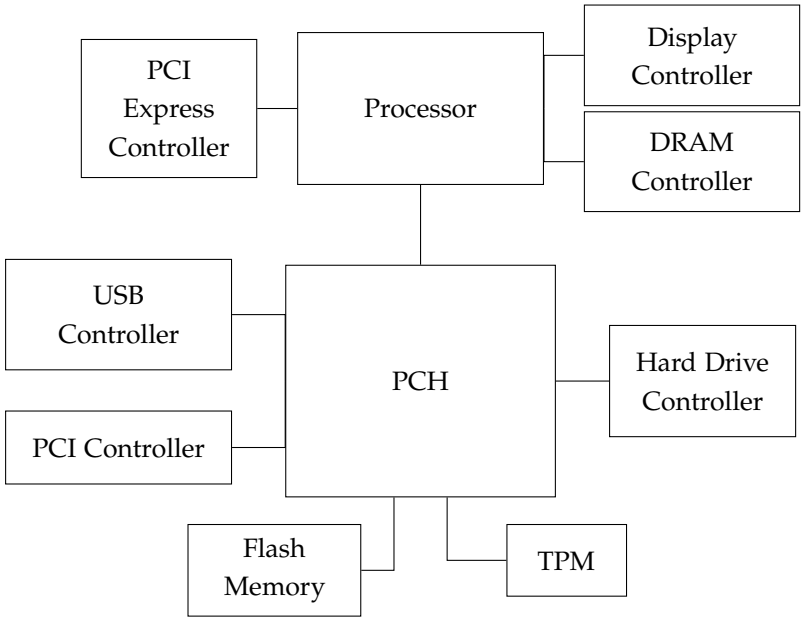


Figure 2.1: High-level view of the x86 hardware architecture

of its documentation. At the time of writing this thesis [1], the *Intel 64 and IA-32 Architectures Software Developer's Manual* is 4,842 pages long. A typical computing platform is made of dozens of hardware components, including *e.g.* hard drives, a keyboard, a trackpad, an audio controller, and a graphic card. Several come with their own documentation, often in the form of large datasheets.

Figure 2.1 pictures how the most important hardware components are interconnected in a modern x86 computing platform (generation *Nehalem* [38] and newer). The two main components are the processor —responsible for executing the software stack— directly connected to high-speed peripherals (*e.g.* DRAM, a display controller), and a companion chipset called the Platform Controller which handles interactions with the rest of the peripherals (*e.g.* USB devices, hard drives). The x86 architecture has seen many iterations over the years, and it is quite common to find in the same computer park different versions of this architecture. For instance, the processor was previously connected to a *northbrige* (low-latency), which was itself connected to a *southbridge* (slower peripherals).

This section proceeds as follows. First, we describe the internals of a x86 processor (2.1.1). We detail the mechanisms which play a part in interactions between the processor and the memories scattered inside the hardware architecture (2.1.2), then focus on the caches embedded inside the processor to reduce the latency induced by these interactions (2.1.3). Finally, we explain how peripherals of the hardware platform actively communicate with the processor (2.1.4).

[1] Spring 2018.

2.1.1 Processor, Architecture and Microarchitecture.

The main component of the architecture is the *processor*. It embeds several execution units called *cores*, which are responsible for executing assembly instructions of software component programs. It also integrates several additional hardware modules to connect the cores to the rest of the hardware architecture, *e.g.* a memory controller to manage interactions with the DRAM. The concrete hardware implementation of the processor is often referred as the Intel microarchitecture, in opposition to the Intel architecture which describes the expected behavior and properties of a x86 system as seen by software developers. While Intel often modifies the microarchitecture, the architecture has remained backward compatible for decades [39]. The microarchitecture implements many optimizations, such as multithreading [40], instruction pipelining [41], out-of-order execution [41, Section 2] or predictive branching [42][41, Section 3].

Intel microarchitecture blurs the frontier between hardware and software. Indeed, an important part of the microarchitecture is not implemented as hardware circuit, but rather under the form of *microcode* programs [31, Section 2.14]. That is, the processor is a programmable device, whose behavior —including the semantics of several x86 instructions it implements [43]— is partly determined by the microcode it has loaded. In practice, x86 processors only load microcode updates which have been signed by Intel. To the best of our knowledge, x86 microcode has never been successfully used as an attack vector.

2.1.2 Memories and Cores I/Os

Besides cores, *memories* are the most important components of a computing platform. Cores interact with these memories during so-called I/Os, for Input/Output: a core receives data during an input and it sends data during an output. The x86 architecture integrates several sources of memories, along with several mechanisms that can be used by the cores to interact with these memories.

The main target of cores I/Os is the Dynamic Random Access Memory (DRAM). DRAM contains the instructions executed by the cores, *and* the data they manipulate. A core decides the semantics of a given memory cell depending on the context, *e.g.* the same binary sequence can be decoded as an instruction or interpreted as an operand of an arithmetic operation. In addition, other hardware components also provide additional memory regions that cores can read from or write to. Contrary to DRAM, I/Os targeting these memories often carry a semantics specific to each peripheral. For instance, an x86 processor integrates a display controller which exposes a frame buffer to the cores. By writing to the frame buffer, a core changes the pictures displayed on the computer screen.

Address Spaces. Cores interact with memories *via* two distinct memory address spaces, characterized by a set of addresses and a set of instructions. The most important address space is the *system memory*, and most of the x86 instructions (*e.g.* `mov` variants, arithmetic operations such as `sub` and `add`) are designed to manipulate it. Addresses of the system memory are referred to as physical addresses and the majority of system memory I/Os are dispatched to the

DRAM. Besides, cores use another address space characterized by two dedicated instructions —`in` and `out`— to target the memories exposed by other hardware components. I/Os issued by `in` and `out` instructions are referred to as Port-Mapped I/O (PMIO), and the addresses of this address space are called *ports*.

Although they are historically used to target different memory regions, nowadays these two address spaces overlap, as peripheral memories and registers can be exposed to the cores *via* the system memory thanks to *memory-mapped* I/Os. That is, *it is possible to read from or write to the same memory location using two different address spaces*. The mapping between addresses of the system memory and their concrete memory locations within the hardware architecture is called the *memory map*. This memory map is configurable, that is it can be changed dynamically *via* configuration registers exposed by the processor and the Platform Controller Hub (PCH).

For instance, the Peripheral Component Interconnect (PCI) standard —whose aim is to propose a standard local bus and communication protocol to connect hardware components to a computing platform— introduces a so-called configuration space per PCI device, which is a dedicated memory region with a specific semantics summarized in Figure 2.2. Registers of the PCI configuration space, such as device and vendor IDs, can be accessed *via* PMIO. Indeed, the PCH exposes two ports to that end. First, software components modify the content of the `PCI_CONFIG_ADDRESS` port, to tell the PCH which PCI configuration register they want to interact with. Then, they read from and write to the `PCI_CONFIG_DATA` port, and the PCH dispatches these I/O to the targeted register. The PCI specification states that the offset `0x10` of the PCI configuration spaces is dedicated to so-called `BAR`s (Base Address Registers). The purpose of the `BAR`s is to configure a memory-mapped mechanism, so that it becomes possible to interact with the PCI configuration space of a given peripheral *via* the system memory. As a consequence, a core which reads from or write to the system memory in a range specified by a BAR sees its I/O dispatched to the configuration space of the related PCI device.

Finally, software components often do not manipulate addresses of the system memory directly. Indeed, cores have their own address translation mechanisms, namely segmentation [34, Volume 3, Section 2.4] and pagination [34, Volume 3, Chapter 4] (potentially extended with its virtualization technology [34, Volume 3, Section 28.2]), which are configurable by the software components.

As such, determining which hardware component will handle a core I/O targeting a given virtual address requires to have a complete knowledge of the x86 remapping and virtual memory mechanisms and of their exact configurations at a given time.

Access Control for Software Components. The x86 hardware architecture provides a rich collection of hardware features [34, Volume 3] to implement fine-grained access control policies about software components I/Os. Subjects of these policies include the software and hardware components of the system. Objects ultimately come down to the memory locations of various natures scattered within the hardware architecture. Actions comprise reading from and writing to a memory location. From a core perspective, it is also common to distinguish between reading data and reading instructions.

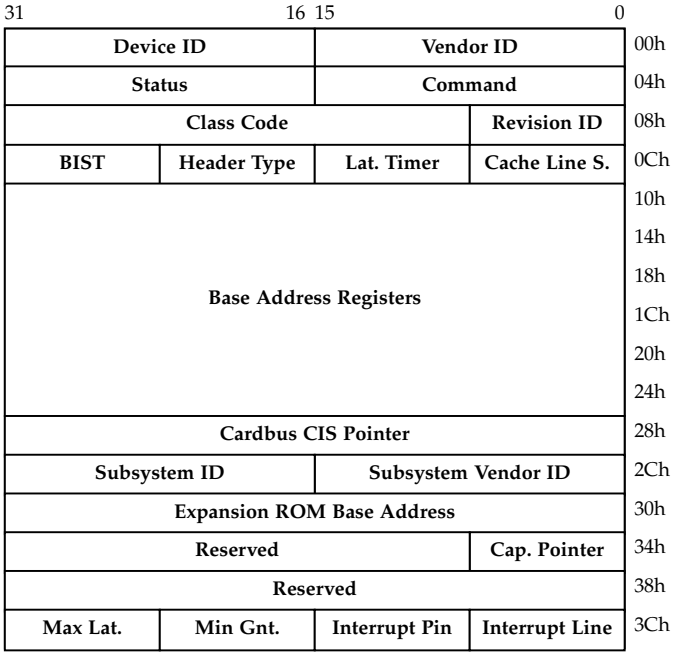


Figure 2.2: Standard registers of PCI Type 0 (Non-Bridge) Configuration Space Header

For instance, the MMU is probably the most well-known hardware feature to implement an access control policy. Thanks to the MMU, an operating system can attribute ranges of DRAM to user applications it manages and isolate its code and data from these applications. The MMU alone is not sufficient, because its scope does not cover its own configuration. That is, it is not possible to configure the MMU in order to prevent a software component to modify the MMU configuration. As a consequence, an additional hardware feature has to be used: the protection rings [34, Volume 3, Section 5.5]. The x86 cores can operate in 4 different so-called rings, from 0 to 3, where ring 0 is the most privileged and ring 3 the least. Ring 3 imposes several restrictions on software components, including the capability to modify the `CR3` register which identifies the base of the page table hierarchy used by the MMU. This is why ring 3 is commonly dedicated to the execution of applications.

The complexity of the x86 I/O resolution mechanism requires to take into account the numerous redirection features exposed by the architecture. Memory locations can have an arbitrary number of aliases, in several layered address spaces: the DRAM controller assigns an address to each memory cell the DRAM contains; the processor maps physical addresses to DRAM addresses; the MMU maps virtual addresses to physical addresses. As a consequence, modifying the content of a memory cell may not be the only way at the disposal of attackers to defeat a given access control policy. For instance, if the access control policy refers to virtual addresses v, modifying the MMU configuration results in modifying the content associated with v.

2.1.3 Cache Memory

Interacting with the DRAM remains slow, in regard to the speed of cores. To improve performance, Intel processors come with several levels of caches, from the smaller and quicker, to the bigger and slower. For instance, Intel Core i7, i5 and i3 processors have three levels of caches. Each core is assigned two levels of cache called L1 —which has the particularity of being divided into a cache of instructions (used when the core read from memory instructions to execute) and a cache of data — and L2, while they all share a so-called L3 cache divided into interconnected slices. Figure 2.3 summarizes this organization.

Caches are divided into several *cache blocks* addressed by an index. Cache blocks are divided into several *cache lines*, which are tagged with a memory address and contains a copy of the data stored at this address. A cache is characterized by the number of cache lines per cache block, the nature of the address associated with cache block indexes, and the nature of the address used to tag cache line. For instance, Intel L2 and L3 caches are *physically-indexed*, *physically-tagged*, meaning they use physical address to compute both the index and the tag.

Caches are mostly transparent to the software components. For instance, the processor alone enforces the cache coherence, with the notable exception of multiprocessor (not multicore) systems, for which "maintenance of cache consistency may, in rare circumstances, require intervention by system software." Intel has developed a dedicated protocol called MESIF to that end [38].

When a core successfully reads the memory at a given address, it keeps a copy of the result in its caches. Therefore, the next time it needs to read some data at this address, these data are retrieved from the cache. Regarding write accesses, Intel x86 processors provide five different caching strategies (uncacheable, write combining, write-through, write-back and write-protected) [34, Volume 3, Chapter 11]. Fine-grained cache strategy configuration is achieved through several hardware mechanisms, including (but not limited to):

- The `CR0` register has a flag called `CD`, which enables caching once set.
- The processor has several registers called Memory Type Range Registers (MTRR), to specify a cache strategy for pre-defined memory regions.
- The Page Table Attribute (PAT) allows for configuring a cache strategy at a memory page granularity.

The write-back strategy is the most commonly used and is summarized in Figure 2.4. The purpose of this strategy is to reduce the number of I/O forwarded to the DRAM. To that end, each cache line has a "dirty bit" which is set by the cache when its value is updated by a write access. A *cache eviction* occurs when the cache frees a given cache line previously used for a given address in order to use it for another one. When it happens, the cache verifies the value of the "dirty bit", so it can update the underlying memory cell if necessary. Therefore, as long as the cache line is not evicted, the processor does not issue write access to the underlying memory. The write-back strategy is not suitable for memories of other hardware components

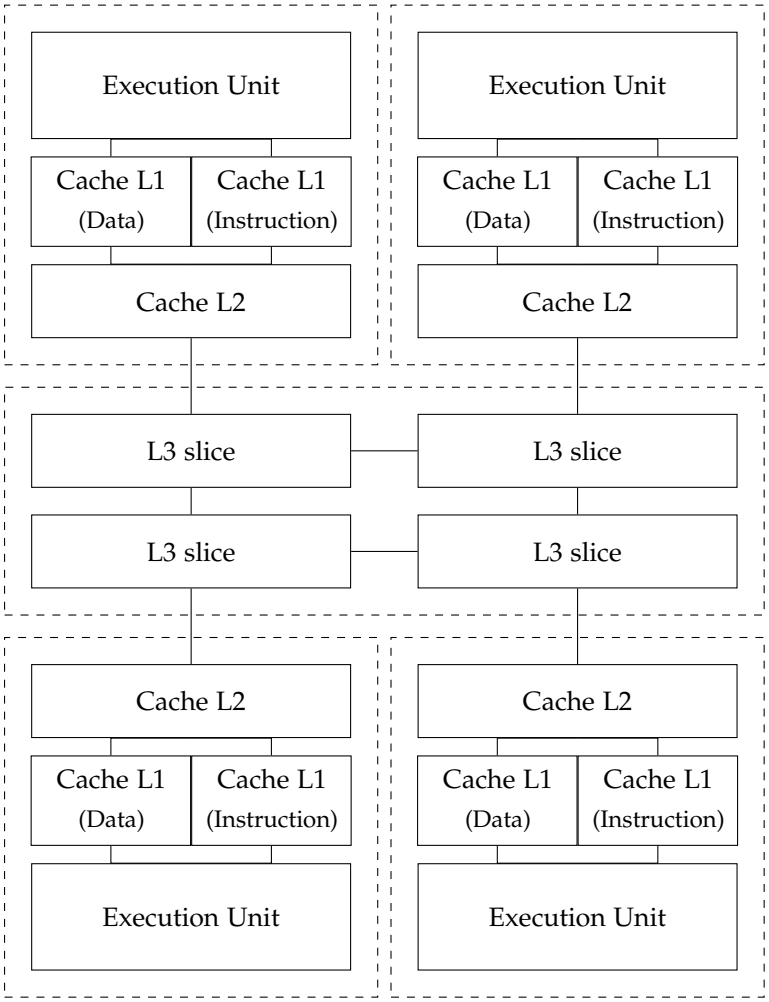


Figure 2.3: Typical caches organization of a x86 processor

mapped into the system memory. For instance, caching I/Os targeting a framebuffer does not make sense, because the screen wouldn't be updated. This is why Intel provides four complementary strategies. The uncacheable strategy disables the cache for the given address, while write combining, write-through, write-back and write-protected are similar strategies such that copies of the underlying memory are stored in the cache to accelerate read accesses, but write accesses are directly forwarded.

2.1.4 Peripherals I/Os

Cores are not the only active hardware components present inside a typical x86 hardware architecture. For instance, several hardware components can also read from or write to the DRAM using a technology called Direct Memory Access (DMA). Hardware components can also interact with the processor by sending hardware interrupts of various natures. When a user presses a key of its keyboard, the latter sends an interrupt request. Interrupt handlers, that is programs executed by the core when it receives interrupts, are configurable *via* a so-called Interrupt Descriptor Table (IDT) [34, Volume 3, Chapter 6]. Each line of the IDT corresponds to a given interrupt whose semantics is specified by Intel, as summarized in Table 2.1. When a core handles an interrupt, it saves its current context inside the DRAM, then starts executing the corresponding interrupt handler. Not all x86 interrupts come from a hardware component. Cores use several of them, for instance to recover from errors. For example, if a core is not able to translate a virtual address into a physical address, it raises a so-called page fault.

Access Control for Hardware Components. Hardware components come from various places and can be of various qualities. Once integrated together, an attacker can potentially leverage any of them to threaten the security of the system. In this context, the principle of least privilege [44] applies: a given component should only be able to leverage capabilities it needs to work according to its purpose, where a "capability" refers to the right to perform a given I/O. To impose an access control policy to hardware components, a system software component uses the so-called VT-d feature, which implements an I/O-MMU for x86 computing platform [45].

In practice, the correct enforcement of an access control policy for hardware components remains challenging, for various reasons. Firstly, security checks can have an important impact on performance. Partly for this reason, the hardware components have long been assumed trustworthy. A good demonstration of this fact is the Address Translation Services mechanism introduced by the PCI standard, whose purpose is to allow PCI devices to bypass security mechanisms designed to reduce their privileges [46]. Other examples showed blind trust in foreign hardware components (*e.g.* USB devices) is not without consequences from a security perspective [47, 48, 49]. Secondly, the "least privilege" may vary from one execution to another, *e.g.* depending on the software stack executed. To handle the numerous use cases of the x86 architecture, its default configuration is very permissive, until software components such as the BIOS or an operating system modify it to fit their needs, by implementing HSE mechanisms.

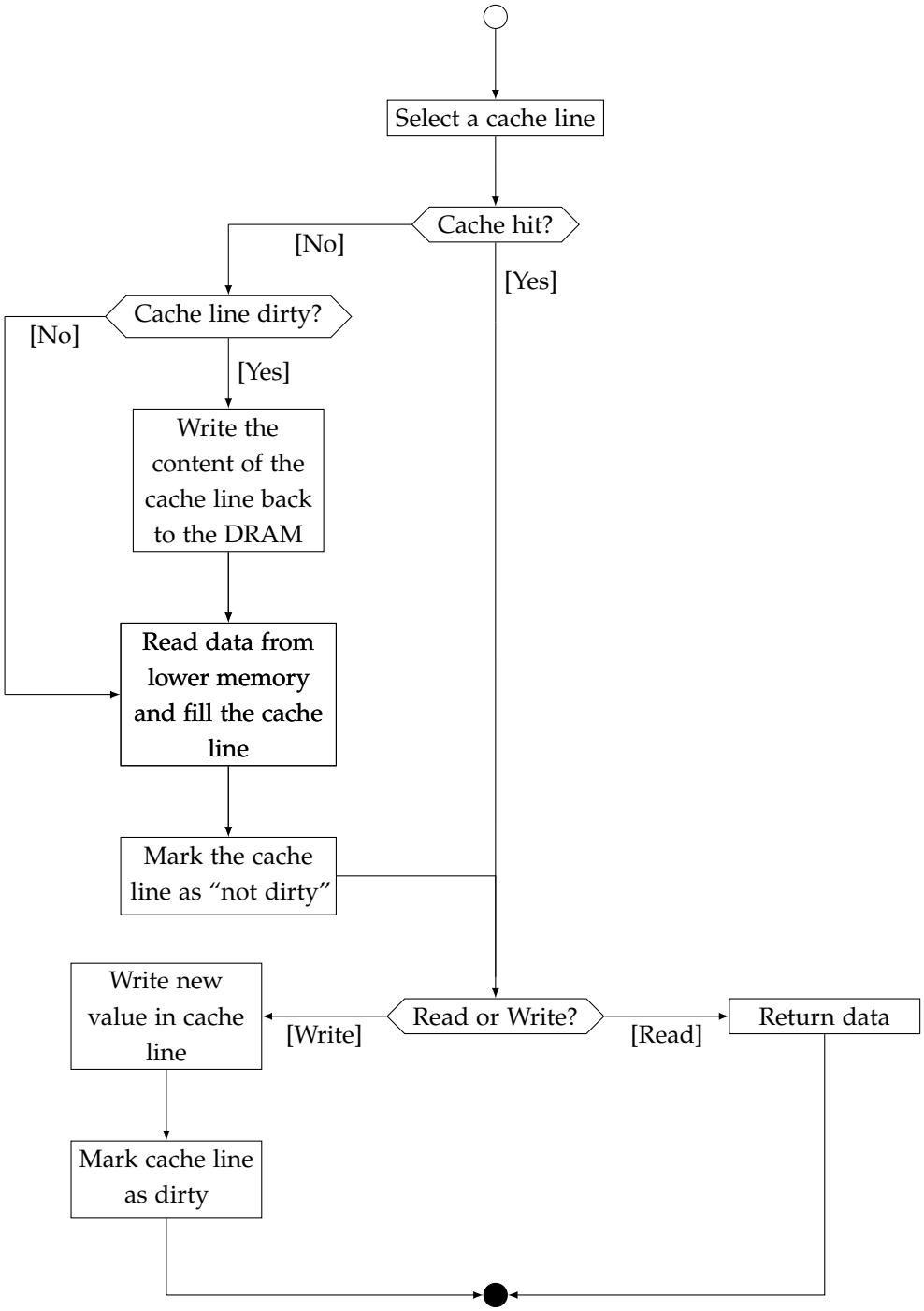


Figure 2.4: The Write-Back cache strategy

#IRQ	Semantics
0x00	Division by zero
0x01	Single-step interrupt
0x02	Non-Maskable Interrupt (NMI)
0x03	Breaking point (used by debuggers)
0x04	Stack overflow
0x05	Bounds
0x06	Invalid instruction opcode
0x07	Coprocessor not available
0x08	Double fault
0x09	Coprocessor segment overrun
0x0A	Invalid task state segment
0x0B	Segment not present
0x0C	Stack fault
0x0D	General protection fault
0x0E	Page fault
0x0F	Reserved by Intel
0x10	Math fault
0x11	Alignment check
0x12	Machine check
0x13	SIMD floating-point exception
0x14	Control protection exception

Table 2.1: x86 Interrupt Descriptor Table semantics

2.1.5 Conclusion

This introduction to the x86 hardware architecture provides the necessary background to understand the challenges related to the implementation of a HSE mechanism in general. In the next section, we explain why the BIOS requires such an isolated environment to operate, and we detailed the hardware features it leveraged to that end.

2.2 BIOS Overview

The BIOS plays a significant role in Intel x86 computing platform. It is the first piece of software executed by the processor, which initializes the hardware components and initiates the execution of the software stack during the boot sequence (2.2.1). At runtime, it remains active to perform various tasks, including and not limited to platform-specific events, device emulation, or BIOS updates management (2.2.2). As such, it can only operate properly if certain security requirements are met and implements several HSE mechanisms to that end (2.2.3).

2.2.1 During the boot sequence

The BIOS program is stored inside a small flash memory connected to the PCH through the Serial Peripheral Interface (SPI) bus on modern x86 computing platform. When the computing platform is powered up, the processor starts executing the code stored at a hard-coded address within the flash memory. The first task of the BIOS is to initialize the hardware architecture [50]. Then, the BIOS searches for a system software component to load into memory. Historically, "legacy" BIOSes were looking for a Master Boot Record (MBR) at the beginning of mass storage devices (*e.g.* hard drive, USB stick). The MBR, whose size is limited to 512 bytes, contains a small program to initiate a loader for a system software component. Modern BIOSes implement the Unified Extensible Firmware Interface (UEFI) [51, 52] standard, whose purpose is to standardize the boot sequence process in order to favor interoperability of BIOS implementations. The boot sequence is divided into several phases, and the BIOS is packaged into several software components accordingly. In particular, UEFI-compliant BIOS can load so-called UEFI applications of arbitrary size, leading modern hypervisors and operating systems to be packaged as UEFI applications [53].

Because the BIOS is the first software component executed by the hardware architecture, and is responsible for initiating the execution of following software components (*e.g.* an operating system), it is commonly designated as the root of trust [54] for the software stack. As such, the integrity of the BIOS code is critical, and several strategies have been proposed to detect BIOS code corruption during the boot sequence, with the two most predominant being Secure Boot [55] and Trusted Boot [56]. Secure Boot and Trusted Boot can uncover certain BIOS corruptions prior to the execution of the illegitimate code. However, they both rely on a so-called *root of trust*, which is the initial code of the BIOS, whose integrity cannot be guaranteed for certain because it is not stored on a read-only memory. Recent efforts have been expended to overcome this limitation. For instance, in 2013 HP has introduced a security mechanism

called SureStart [57] whose purpose is to move the root of trust within another hardware component, leaving most attackers unable to modify its code. More recently, the NIST has published the Special Communication 800-193 —*Platform Firmware Resiliency Guidelines* [58]— which specifically tackles the challenge posed by illegitimate firmware modification. These approaches have in common to aim at enforcing the correct initialization of the platform by the BIOS.

2.2.2 At runtime

The boot sequence ends once a system software component has been selected and loaded into memory by the BIOS.

Software Interfaces. At runtime, the BIOS provides various software interfaces to the system software component. For instance, the Advanced Configuration and Power Interface (ACPI) tables [59, 60] is a standardized interface to configure various vendor-specific aspects of the hardware platform, such as power management or thermal management. Similarly, legacy BIOSes expose facilities to system software components, in the form of so-called BIOS Interrupt. For instance, the interrupt `0x10` is dedicated to video services (*e.g.* setting the video mode, setting the cursor shape and position, etc.). Nowadays, UEFI-capable BIOSes expose so-called *Runtime Services* to system software component [61, Chapter 5] under the form of a table of function pointers.

In either case, these interfaces act as an intermediary layer between a system software component and the hardware architecture. In doing so, they reduce the coupling between the software and hardware components. Sometimes, their use is optional, and BIOSes only provide them as a facility. Other are mandatory gates towards certain computing platform features, because they are related to critical mechanisms of the platform and the hardware vendors do not want to rely on a (potentially vulnerable or malicious) system software component. For instance, the BIOS takes care of its own software updates, in order to verify submitted versions prior to applying them, *e.g.* by verifying cryptographic signature or preventing the installation of older, outdated versions.

Proactive Features. In addition to supporting the execution of the rest of the software stack through its interfaces, the BIOS carries out several hardware-specific tasks which are not publicly documented. This includes and is not limited to handling hardware errors, checking thermal zones, adjusting cores speed, configuring hardware workarounds, and emulate complete hardware devices to the system software component [62].

The execution of the BIOS in this context should be transparent to the rest of the software stack. As such, the BIOS remains the most privileged software component of the software stack, even after the end of the boot sequence.

2.2.3 HSE Mechanisms Implemented by the BIOS

The BIOS is provided by the manufacturer of the hardware architecture. In most cases, it is a proprietary software, and the computer owner has little control over it. The rest of the software stack is considered untrusted, and one goal of the BIOS is to keep the computer in a working state, even in the presence of an erroneous or malicious software stack. To that end, the BIOS relies on several HSE mechanisms to enforce its isolation from the rest of the software stack. From the information in the Intel manual [34], datasheets [35, 36], and in the academic literature [27], the isolation required by the BIOS can be divided into three complementary security policies.

Volatile Memory Access Control The BIOS is assigned a region of the volatile memory to support its execution at runtime. This region is protected against I/Os issued by the rest of the software stack.

Availability The rest of the software stack is not authorized to prevent the execution of the BIOS at runtime, *i.e.* the BIOS can preempt the execution of the rest of the software stack.

Non-volatile Memory Access Control The BIOS is assigned a non-volatile memory region (in practice, a portion of the flash memory) to store its code and data. The rest of the software stack is not authorized to modify the content of this memory, in order to avoid a scenario where attackers modify the BIOS code according to their needs and provoke a reboot of the platform.

These three security policies are implemented by the means of HSE mechanisms which rely on hardware features exposed by the processor and the PCH.

System Management Mode. To handle several software components with different levels of privilege, Intel processors provide several execution modes, which can be assimilated to sets of hardware capabilities. For instance, in a given execution mode, a core may refuse to execute certain assembly instructions. Contrary to common belief, x86 execution modes are not organized in a linear hierarchy, but are rather a matrix of complementary hardware features: protection rings, paging configuration, virtualization technologies, etc. As for the BIOS, Intel provides the so-called System Management Mode (SMM) [34, Volume 3, Chapter 34], introduced in the Intel manual as follows:

> SMM is a special purpose operating mode provided for handling systemwide functions like power management, system hardware control, or proprietary OEM-designed code. It is intended for use only by system firmware, not by application software or general-purpose system software. The main benefit of SMM is that it offers a distinct and easily isolated processor environment that operates transparently to the operating system or executive and software applications.
>
> *Intel 64 and IA-32 Architectures Software Developer's Manual*

The SMM is the foundation of the BIOS isolation at runtime, but it is not sufficient.

System Management RAM. The SMRAM is the name given by Intel to a memory region located inside the DRAM, and dedicated to the SMM. The exact location and size of the SMRAM are architecture dependent. To locate it, the processor uses a dedicated register named `SMBASE`. The BIOS should configure it during the boot sequence. As its name suggests, the `SMBASE` value should point to the base of the SMRAM. As for the end of the SMRAM, the hardware architecture does not expose it explicitly, and the BIOS developers need to refer to the processor datasheet in order to find it.

At the beginning of the boot sequence, the SMRAM is left unprotected, meaning arbitrary memory accesses targeting the SMRAM are authorized. This design allows the BIOS to initialize the SMRAM content. Once the SMM code —the BIOS code intended to be executed at runtime in SMM— has been correctly loaded into the SMRAM, and prior to starting the execution of a system software component, the BIOS has to lock the SMRAM. A locked SMRAM can only be accessed by a processor in SMM. To that end, the memory map is dynamically modified with respect to the current state of the processor, and the physical addresses dedicated to the SMRAM from the BIOS perspective are used by the rest of the software stack to access the VGA controller memory. The `SMRAMC` register, exposed by the processor, controls this access control mechanism *via* its `D_LCK` bit. The BIOS locks the SMRAM by setting the `D_LCK` bit. The memory controller of a processor with the `D_LCK` bit set will prevent I/Os targeting the SMRAM if the processor is not in SMM. In addition, the only way to clear the `D_LCK` bit is by performing a complete reboot of the platform. This leaves no opportunity for the rest of the software stack to modify the content of the SMRAM, because the BIOS will lock it again during the boot sequence, prior to executing another software component.

System Management Interrupt. The System Management Interrupt (SMI) is a hardware interrupt which makes cores "enter" SMM. More precisely, when a core receives a SMI, it saves its current state (*e.g.* its registers, current execution mode, etc.) in the SMRAM, then it reconfigures itself; in particular, it sets its program counter register to the value `SMBASE` + 0x8000. From this point, the core is in SMM and starts to execute what should be the SMM code. Once the SMM code has performed the task it has been requested for, the `rsm` instruction can be used. This instruction, specific to the SMM, tells the core to exit SMM and to restore its previous state. This way, the execution of the software component previously halted by the SMI can resume. From the system point of view, it is almost like if nothing has happened.

Finally, the PCH exposes a register called `APM_CNT` that a system software component can write to in order to make the PCH trigger a SMI [36]. In practice, this mechanism is used by a system software component in order to request the execution of the BIOS, for the purpose of carrying out a given service, *e.g.* modifying the content of a given UEFI variable. [2]

Flash Memory Lockdown. The content of the flash memory has to be protected from arbitrary write accesses, similarly to the SMRAM protection. That is, only the SMM code should be able to overwrite the content of the flash memory. This access control mechanism is implemented by

[2] UEFI variables are stored in the flash memory, alongside the BIOS code.

the PCH, and is configurable *via* the BIOS_CNTL control register. Two bits of this register are of interest: the BIOSWE (BIOS Write Enable) bit, and the BLE (BIOS Lock Enable) bit.

The semantics of the BIOSWE and BLE bits is as follows. When the BIOSWE bit is clear, the PCH only authorizes read accesses to the flash memory. If a core sets the BIOSWE bit, the behavior of the PCH depends on the value of the BLE bit. If the BLE bit has been set by the BIOS during the boot sequence, then the PCH triggers a SMI. As a consequence, the cores stop their current executions and enter in SMM. This prevents a system software component from modifying the content of the flash memory, even if the PCH now authorizes write accesses. It is the SMM code responsibility to clear the BIOSWE bit before using the rsm instruction. On the contrary, if the BLE bit is not set, setting the BIOSWE bit will not cause a SMI, leaving the system software component free to modify the content of the flash memory. Similarly to the D_LCK bit of the SMRAMC register, the BLE bit cannot be cleared without a reboot.

The system software component and the BIOS often use the flash memory lockdown mechanism as a communication channel. The system software sets the BIOSWE bit in order to notify the SMM code that a BIOS update is available.

The combination of the SMRAM, the SMI and the flash memory lockdown explains why the SMM is often referred to as the x86 "most privileged execution mode." In a nutshell, the SMM code can leverage the same hardware capabilities as the system software, including manipulating memories used by the system software. On the contrary, the system software cannot modify either the SMRAM content or the SMM code stored in the flash memory, and cannot prevent the SMM code execution, *i.e* intercept or mask SMIs.

2.3 BIOS HSE Mechanism and Compositional Attacks

To stay isolated from the rest of the software stack at runtime, the BIOS implements a HSE mechanism whose key hardware feature is the SMM, a dedicated execution mode of x86 processor. The SMM provides the necessary features to enforce its isolation from the rest of the software stack. Despite the key importance of SMM, several compositional attacks have been disclosed over the past decade. In this section, we detail three attacks which have defeated the three security policies targeted by the BIOS. The SMRAM Cache Poisoning Attack allowed for circumventing the SMRAM access control mechanism (2.3.1). The so-called SENTER Sandman attack prevented the execution of the BIOS and led to modifying the content of the flash memory (2.3.3). The Speed Racer attack resulted into an authorized modification of the flash memory as well (2.3.2).

2.3.1 SMRAM Cache Poisoning Attack

Between 1986, when the SMM has first been introduced, and 2009, it was believed that the SMRAMC register alone was sufficient to enforce SMRAM access control. Loïc Duflot *et al.* [4]

and Rafal Wojtczuk *et al.* [5] independently showed that this belief was misplaced when they disclosed the SMRAM Cache Poisoning Attack.

Attack Path. The SMRAM Cache Poisoning leverages the write-back strategy (2.1.3) of the cache to circumvent the `D_LCK` bit protection. The attack proceeds as follows:

1. Attackers set the cache strategy to be used for the SMRAM addresses to write-back. This can be done by a malicious system software component, or even by a malicious application under certain circumstances (for instance, if an operating system exposes a software interface to manage the cache from the userland).

2. They write to the `APM_CNT` register in order to trigger of a SMI.

3. The BIOS code stored in SMRAM is executed in SMM, leading the cache to be filled with copies of that code, and the processor leaves SMM when it executes the `rsm` instruction.

4. Attackers write to an address which belongs to the SMRAM, and because of the write-back cache strategy, the processor updates the copies within the cache, and does not forward the I/O to the memory controller.

5. Attacker trigger another SMI, and the processor uses the modified copy of the SMM code inside its cache.

This attack is a perfect illustration of a compositional attack: both the memory controller and the cache work as expected. The former prevents authorized accesses to the SMRAM, that is a subset of the DRAM, by a processor not in SMM; the latter is keeping copies of successful accesses to decrease latency due to memory accesses. However, the composition of the cache and the memory controller breaks the BIOS Integrity property.

Countermeasure. The solution implemented by Intel to prevent further exploitation of this vulnerability was to modify the behavior of the cache, when some memory access target the SMRAM. Because the SMRAM size and location remain specific to each architecture, this means it requires an additional step of configurations to tell the cache the physical addresses that belong to the SMRAM.

The SMRAM Cache Poisoning attack is a textbook case of compositional attacks. It is interesting to notice that six years later, Christopher Domas has disclosed another x86 vulnerability called Sinkhole [6], which relies on a similar approach —but different hardware features— to trick a processor in SMM to execute arbitrary instructions. Both attacks leave the content of the SMRAM in DRAM intact, and leverage only legitimate hardware features.

2.3.2 Speed Racer

In 2015, Corey Kallenberg *et al.* showed that the scenario detailed previously, such that setting `BIOSWE` triggers a SMI to suspend the execution of the system software, suffered from a race condition if two cores cooperate [7].

Attack Path. On a typical x86 hardware architecture, all the x86 cores of the platform will *eventually* enter SMM when a SMI is triggered. On the contrary, the `BIOSWE` flag is set as soon as the `BIOS_CNTL` register is modified. If two cores cooperate, they can benefit from a sufficient window for action and successfully tamper with the flash memory content. The attack proceeds as follows:

1. One core tries *ad infinitum* to overwrite the content of the flash memory. Because the `BIOSWE` bit is initially clear, the PCH discards its attempts, and the flash memory content is correctly protected.
2. At the same time, another core set the `BIOSWE` bit.
3. A SMI is triggered, but by the time it propagates to the first core, it may have successfully modified the flash memory content.

Countermeasure. To prevent this race condition, Intel has introduced a new configuration bit to the `BIOS_CNTL` register: the `SMM_BWP` (SMM BIOS Write Protection). If the `SMM_BWP` is set, the PCH discards any write access which targets the flash memory *unless all processors are in SMM*.

2.3.3 SENTER Sandman

Another attack has defeated the flash memory lockdown protection. In 2015, Xeno Kovah *et al.* showed it was possible to leverage the Intel TXT technology to circumvent the flash memory lockdown protection [1].

Intel TXT. The Intel Trusted eXecution Technology (TXT)[2] is a feature of some x86 processor, whose purpose is twofold: it attests the integrity of the system software component program without the need to trust the BIOS, and it provides a trusted execution environment to the system software component.

Attack Path. The flash memory lockdown mechanism was based on the assumption that unlocking the flash memory would force the execution of the BIOS, so that the latter could lock it again. To that end, the PCH triggers a SMI at the same time as it unlocks the flash memory. This mechanism was introduced at a time when software components could not configure x86 processors to ignore SMIs. This assumption became incorrect when Intel introduced the first version of TXT, whose "trusted execution environment" provided by TXT-capable processors explicitly disabled SMIs handling. As a consequence, adversarial system software components

whose execution were initiated with TXT were able to unlock the flash memory, without being interrupted by the SMI triggered by the PCH in response. Then, they could freely modify the content of the flash memory, left unprotected by the PCH.

Countermeasure. The `SMM_BWP` configuration bit makes this attack ineffective. Besides, recent x86 processors do not disable SMI during the initialization of a system software component with TXT anymore.

2.4 Conclusion

In this Chapter, we gave an overview of the x86 hardware architecture and of the role played by the BIOS within the software stack. We then have detailed how the BIOS relies on several hardware features to remain isolated from the rest of the software stack. This illustrates real life HSE mechanisms. Finally, we have presented three compositional attacks, to better illustrate the threats they pose.

The rest of this manuscript will use the SMRAM Cache Poisoning Attack as a recurring application use case for our contributions, because it has motivated our desire to formally specify and verify HSE mechanisms, and it is a good illustration of compositional attacks.

3

State of the Art

"We build our computers the way we build our cities: over time, without a plan, on top of ruins."

— Ellen Ullman

The formal verification of a system consists of proving its correctness with respect to a *specification* [63], that is a description of properties targeted by the system. To that end, a model of the system is defined to enable rigorous reasoning on the system behavior by means of formal methods.

In this thesis, our objective is to give a formal theory of HSE mechanisms in the form of requirements that trusted software components have to satisfy, and to verify that these requirements do effectively provide the enforcement of targeted security policies. We steer a middle course between two domains: hardware verification and system software verification. Generally, hardware verification focus on properties which are transparent to the executed software (*e.g.* cache coherency [8], linearizability of SGX instructions [64], or hardware-based memory isolation [65]), and system software verification relies on hardware models which abstract as much as possible of the architecture complexity. To the extent of our knowledge, the closest related research project is the work of Jomaa *et al.* [11], who specified what requirements a microkernel must satisfy in order for a MMU to enforce memory isolation at all times. This approach remains an isolated initiative, hence the interactions of the numerous configurable hardware features is less subject to formal verification.

The rest of this Chapter proceeds as follows. We detail how transition systems have been widely and successfully used in order to model and verify hardware and software systems, and where our approach stands with respect to previous work (Section 3.1). Then, we introduce and justify our interest in compositional verification approaches that enable the "divide and conquer" strategy to reduce the burden of verifying large systems (Section 3.2).

3.1 Towards the Formal Verification of HSE Mechanisms

In a verification problem, the definition of a model allows for unambiguously describing the set of possible behaviors of the system. The system is characterized by a set of *states* and by a set of state transformations, called *transitions* (3.1.1). A sequence of transitions of the model, commonly called *trace*, describes how the system has operated over time, that is which behavior it has exhibited. As a consequence, by reasoning about the set of traces of the model, we reason about the set of all possible behaviors of the system, *e.g.* to verify that security policies are enforced at all time (3.1.2). To support this reasoning, many approaches and tools have been proposed. Choosing between one of these tools means making a necessary compromise between expressiveness, automation, and applicability (3.1.3). Regarding our objective, we need to use hardware models which take into account the composition of the many hardware components involved in HSE mechanisms implementation. Many x86 models have been proposed over the years, but to the extent of our knowledge, none of them cover the hardware features we are primarily interested in (3.1.4). To experiment with our formal theory of HSE mechanisms, we propose our own model, and use this opportunity to identify necessary properties that a large-scale model should exhibit in order to remain applicable during a verification process.

3.1.1 Modeling a Hardware Architecture

A system is characterized by a set of states (*e.g.* the state of a processor core typically includes the values of its registers and its execution mode) and a set of state transformations (*e.g.* the value of a register will change when a core executes certain instructions) which occur over times as the system operates. Formally defining these two sets means constructing a model of the system. Labeled transition systems [66] form a prominent class of transition systems, characterized by the use of labels to distinguish between transitions.

Definition 3.1 (Labeled Transition System)
A labeled transition system is a tuple $\langle S, L, R \rangle$, such that S is a set of states, L is a set of labels, and $R \subseteq S \times L \times S$ is the transition relation, that is $(s, l, s') \in R$ is a transition of the LTS.

Different names have been used to designate labels and labeled transitions, *e.g.* input variables [67], operations [68], rules [69], or actions [70]. Although the name changes, the motivation remains to characterize the nature of the transition. LTS are well suited to reason about the interactions of a system with its environment, *e.g.* hypercalls handled by a hypervisor [70] or I/Os provoked by the execution of machine instructions [65]. Similarly to the work of Lie *et al.* [65], we use labels to reason about the interactions between the hardware architecture and the software components that are executed by this architecture. It allows us to define requirements trusted software components have to satisfy to implement a given HSE mechanism.

Throughout this Chapter, we will use the airlock system as a running example, since it is both simple —our examples remain of manageable size— and rich —we can use it to illustrate the definitions we introduce. An airlock system is a device made of two doors, and an intermediary

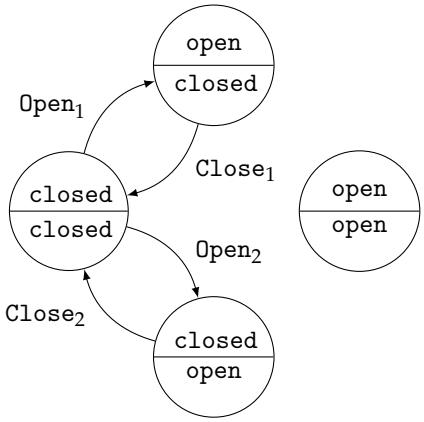


Figure 3.1: A simple airlock system modeled as a labeled transition system

chamber. To get across an airlock system, a user requests the opening of the first door, enters the chamber, waits for the system to close the first door and open the second door, and exits the chamber.

Example 3.1 (Airlock System)
We model our airlock system with a labeled transition system $\langle S, L, R \rangle$, such that:

– A door of the system can be either `open` or `closed`. The set of states of the airlock system reflects the Cartesian product of the doors states.

$$S \triangleq \{\texttt{open}, \texttt{closed}\} \times \{\texttt{open}, \texttt{closed}\}$$

– A transition is characterized by a request to open ($\texttt{Open}_i$, with $i \in \{1, 2\}$) or close ($\texttt{Close}_i$, with $i \in \{1, 2\}$) a door of the system.

$$L \triangleq \{\texttt{Open}_1, \texttt{Close}_1, \texttt{Open}_2, \texttt{Close}_2\}$$

– The model does not allow the simultaneous opening of both doors, as stated by the definition of R which does not contain a transition which leads to the state $(\texttt{open}, \texttt{open})$.

$$\begin{aligned} R \triangleq \{&(\texttt{closed}, \texttt{closed}), \texttt{Open}_1, (\texttt{open}, \texttt{closed}), \\ &(\texttt{closed}, \texttt{closed}), \texttt{Open}_2, (\texttt{closed}, \texttt{open}), \\ &(\texttt{open}, \texttt{closed}), \texttt{Close}_1, (\texttt{closed}, \texttt{closed}), \\ &(\texttt{closed}, \texttt{open}), \texttt{Close}_2, (\texttt{closed}, \texttt{closed})\} \end{aligned}$$

The resulting labeled transition system is pictured in Figure 3.1.

Our objective is to verify HSE mechanisms, therefore we need to be able to model them. To that end, it is important to consider that (1) hardware architectures often allow for implementing

several HSE mechanisms, and (2) hardware features involved in HSE mechanisms are not safe by default. By defining HSE mechanisms of a given hardware architecture against a general-purpose model of this architecture, rather than relying on ad-hoc models dedicated to specific HSE mechanisms, we believe we reduce the overall verification effort induced by (1), and address the threat posed by compositional attacks. However, (2) means such a general-purpose model of hardware architectures necessarily embeds behavior which are legitimate with respect to the hardware architecture functional specification, but violate some targeted security policies. This is the case, for instance, if the BIOS leaves the SMRAM unlocked at the end of the boot sequence.

3.1.2 Specifying Security Policies

The theory of properties of a transition system is now well understood, with an intuitive classification of properties, such that:

– *Safety properties* [17, 18] characterize that nothing "bad" shall *never* happen.

– *Liveness properties* [18, 19] characterize that something "good" shall *eventually* happen.

Two classes of security policies commonly targeted by x86 HSE mechanisms are access control and availability policies. An access control policy is a safety property: unauthorized action by a subject shall never happen. An availability policy is a liveness property: the system shall eventually satisfy the service.

Safety and liveness properties are expressed against sequences of transitions, commonly called *traces* in the literature. Each formalism has its own definition of traces, which takes into account its characteristic. As for the labeled transition systems, their traces interleaves a label between each state [9].

Example 3.2 (Airlock System Trace)
With $\mathtt{c} = \mathtt{closed}$ and $\mathtt{o} = \mathtt{open}$, the crossing, by a user, of the airlock system is characterized by the following trace:

$$(\mathtt{c},\mathtt{c}) \xrightarrow{\mathtt{Open}_1} (\mathtt{o},\mathtt{c}) \xrightarrow{\mathtt{Close}_1} (\mathtt{c},\mathtt{c}) \xrightarrow{\mathtt{Open}_2} (\mathtt{c},\mathtt{o}) \xrightarrow{\mathtt{Close}_2} (\mathtt{c},\mathtt{c})$$

Afterwards, we write $\Sigma(M)$ for the set of traces of a labeled transition system M.

Safety and liveness properties can be defined in terms of predicates on traces [71, 72, 73]. M is said to be correct with respect to a property modeled as a predicate on traces P when

$$\forall \rho \in \Sigma(M), P(\rho).$$

On the one hand, safety properties are characterized by an invariant ι on trace elements, that is

$$P(\rho) \triangleq \iota(\rho_0) \wedge P(\rho_{[1..]}),$$

where ρ_0 is the first element of the trace, and $\rho_{[1..]}$ is the trace obtained by removing the first element of ρ. On the other hand, liveness properties are characterized by a predicate η on trace which has to be satisfied for at least a subtrace, that is

$$P(\rho) \triangleq \exists n > 0, \eta(\rho_{[..n]}) \vee P(\rho_{[1..]}),$$

where $\rho_{[..n]}$ is the subtrace made with the n first elements of ρ.

Example 3.3 (Airlock Safety and Liveness Properties)
A typical *safety* property for an airlock system is that at least one door shall be closed at any time. We formalize this property with the invariant ι, defined as follows:

$$\iota(d_1, d_2) \triangleq d_1 = \texttt{closed} \vee d_2 = \texttt{closed}$$

The specification of the airlock system is defined with a labeled transition system. Assuming the airlock device is initialized in a correct state (*e.g.* both doors are closed), we verify this specification is correct with respect to the safety property characterized by ι by exhibiting a proof that ι is an invariant with respect to R, that is

$$\forall((d_1, d_2), l, (d_1', d_2')) \in R, \iota(d_1, d_2) \Rightarrow \iota(d_1', d_2')$$

Proof. By definition of R, there are four transitions to consider, and our proof consists in an enumeration of these cases. Case 1, both doors are closed, that is $d_1 = d_2 = \texttt{closed}$. The system opens the first door, that is $l = \texttt{Open}_1$. At the end of the transition, only the first door is open, that is $d_1' = \texttt{open}$ and $d_2' = \texttt{closed}$. By definition of ι, the statement to prove becomes

$$\texttt{closed} = \texttt{closed} \vee \texttt{closed} = \texttt{closed} \Rightarrow \texttt{open} = \texttt{closed} \vee \texttt{closed} = \texttt{closed}$$

By definition of the disjunction $\vee$, the statement reduces to True $\Rightarrow$ True, which is True by definition of the implication $\Rightarrow$. The three other cases follow the exact same procedure. □

In addition, we can also prove that both doors of the airlock will *eventually* be closed. We can characterize this liveness property with the predicate η on subtraces of one element, such that

$$\eta(d_1, d_2) \triangleq d_1 = \texttt{closed} \wedge d_2 = \texttt{closed}$$

We verify the specification of the airlock system is correct with respect to the liveness property characterized by η by exhibiting a proof that for each transition of R, one of the states satisfies η, that is

$$\forall((d_1, d_2), l, (d_1', d_2')) \in R, \eta(d_1, d_2) \vee \eta(d_1', d_2')$$

Proof. Again, the proof proceeds by case enumeration with respect to the definition of R, and after unfolding η, we conclude using the definition of $\wedge$ and $\vee$. □

Not all security policies can be formalized with predicates on traces. For instance, *noninterference* [74] is a confidentiality policy which requires that so-called public inputs handled by a

given system always produce the same output, regardless of concurrent secret inputs. In this context, considering each trace independently is not sufficient, as to witness a violation of the security policy requires to compare two traces together. As a consequence, such policies are characterized by sets of sets of traces; they are called *hyperproperties* [40]. Verifying a system with respect to a hyperproperty is harder in the general case, but certain hyperproperties, called k-safety properties [40], can be reduced to an invariant enforcement which is more manageable to prove.

In the context of formal verification, a security policy is characterized by a (set of) set(s) of traces wherein the policy is enforced at all times. Because hardware features are unsafe, *i.e.* they need to be configured by trusted software components which implements HSE mechanisms, legitimate traces of a general-purpose x86 model will not be part of the "secure" traces. Our formal theory of HSE mechanisms allows for identifying the subset of traces wherein the mechanism is correctly implemented by the trusted software components. Verifying a HSE mechanism consists of proving that this set of traces satisfies the predicate which characterizes the targeted security policy. Because of the complexity of a typical x86 architecture, this work cannot be achieved manually and we need the support of an appropriate tooling.

3.1.3 Approaches and Tools

Formal verification leverages two classes of approaches with their dedicated tools. On the one hand, the most generic approach is the construction of a proof, that is a succession of inference rules to derive the statement to be proven from formulas known to be true. Theorem provers provide facilities to construct proofs and a checker to automatically verify these proofs. On the other hand, it is possible to rely on algorithms whose correctness has been formally established to handle a well-defined class of problems. Model checking [75] and satisfiability solvers [76] are two classical instances of this approach.

The choice of one approach over another for a given verification problem is a compromise constrained by requirements in terms of expressiveness, automation and applicability. More expressiveness for formal language increase the scope of verification problems it can cover, but it also reduces the level of automation of the tools used to verify its statements. Fully automated procedures rely on algorithms whose computational complexity impairs their applicability for more complex problems, *e.g.* model checking and the *state explosion problem* [77].

We now give an overview of state-of-the-art tools which are commonly leveraged for formal verification of hardware and software components. We have organized it according to the formal language they rely on.

Propositional and First-Order Logic. Propositional logic is characterized by *terms*, which represents objects, and *logical operators*, such as conjunction $\wedge$, disjunction $\vee$, implication $\Rightarrow$, and negation $\neg$. First-order logic [78] extends propositional logic with *predicates*, that is parameterized formulas that can be true or false with respect to the applied terms, and *quantifiers*, such as $\forall$ (all values) or $\exists$ (there exists a value).

SAT solvers [79] (*e.g.* Chaff [80])) form a class of tools whose purpose is to determine whether a given propositional formula with boolean variables is satisfiable, by finding appropriate instances for these variables. SMT solvers [81] (*e.g.* Z3 [82]) have a similar purpose, but they target first-order logic formula —without quantifiers— and are not limited to boolean variables (SMT stands for satisfiability modulo theories). Hence, in the theory of natural numbers, an SMT solver will find that the formula $x > y \wedge x + y = 5$ is satisfiable with $\{x = 3, y = 2\}$, but the formula $x - y = 0 \wedge x \neq y$ is not satisfiable by any assignment. The lack of quantifiers has a significant impact on formula expressiveness which reduces its applicability when it comes to formal verification, *e.g.* we cannot express a specification of the form "at any point in a trace, a predicate P is satisfied" if we consider a system can operate infinitely.

Temporal Logic. Modal logic [83] is a formal system which extends first-order logic with modal operators. Temporal logic forms a family of modal logic systems, including *e.g.* Linear Temporal Logic (LTL) [84] and Computation Tree Logic (CTL) [85]. Examples of LTL modular operators are $\Box P$ (P is always true), $\Diamond P$ (P will eventually become true), and $\bigcirc P$ (P will be true after the next transition of the system). CTL considers trees of possible futures (in opposition to a *linear* future). CTL modal operator includes **A**P (P is true for all possible futures) and **E**P (there exists at least one path where P becomes true). Temporal logic operators allow for reasoning about propositions over time on infinite traces. Temporal logic formulas are commonly verified using model checkers, *e.g.* NuSMV [67], SPIN [86] or TLA+ [87].

Despite improvements in algorithms, model checkers remains subject to the *state explosion problem* [77], whose mitigation reduces the automation of the approach. For instance, symbolic model checking requires to manually provide invariants to act as inductive hypotheses [88], and bounded model checkers require to fix an upper bound for traces size, with the risk to abate the result in the chosen number is too low [89].

Higher-order Logic. First-order logic quantifiers can only be applied on sets of terms (*e.g.* natural numbers, booleans, or more complex structures such as the states of an airlock system). Higher-order logic [90] does not suffer the same limitation, since they allow quantification over sets of sets, functions and predicates. Hence, it becomes possible to express statements such as *for all sets with a total order* $<$, *for all pair of values* α, β, *then either* $\alpha < \beta$ *or* $\beta < \alpha$ *or* $\alpha = \beta$. Therefore, higher order logic is more expressive than first-order logic, but it comes at a cost in terms of automation. Interactive theorem provers (*e.g.* Coq [22], Isabelle/HOL [91], Atelier B [92], or more recently Lean [93]) are based on higher-order logic.

We leverage a higher-order logic to define our formal theory of HSE mechanisms. This allows us to reason about high-level properties implying HSE mechanisms —*e.g.* safety property enforcement (Theorem 4.1) or mechanisms composition (Theorem 4.2)— where the hardware model is left as a parameter of the proof.

3.1.4 Tour of Existing x86 Models

To experiment with our approach, we decided to focus on the HSE mechanism implemented by the BIOS at runtime to remain isolated from the rest of the software stack (Section 2.3). We specify this mechanism and verify its correctness with respect to the security policy it aims to enforce using Coq. We now give an overview of existing x86 models, and justify our choice to develop our own model.

x86 Models by Intel. Intel has integrated formal verification in its processor design process for more than a decade. Intel engineers first verified arithmetic operations performed by the processors [94], then increased the verification scope to cover the complete execution unit [95]. More recently, the SGX instructions —which allow system software components to create and manage enclaves [31]— have been verified with respect to security properties, rather than functional specification [64].

Each project focused on one aspect of the x86 architecture and led to uncover logic errors and inconsistencies in processor designs, while we aim for a more general-purpose model which takes into account that hardware architecture are made of many components which interact together. These (sometimes unsuspected) interactions may pave the road towards compositional attacks. To the extend of our knowledge, Intel has not advertised about a "global" model of its architecture, even though Nachiketh Potlapally, who was working at Intel at the time, discussed the benefits of such a model in 2011 [96].

Besides, the models and tools used by Intel are rarely publicly available, hence we could not leverage them to experiment with our approach.

x86 ISA Models. Several x86 model have been proposed by academic researchers over the years for purposes of formally reasoning about machine-code programs. To that end, they have modeled the x86 instructions set semantics, often referred to as x86 ISA.

Probably the most mature projects include RockSalt by Morissett *et al.* [97], Goel *et al.* framework [98], and CompCert x86 assembly model [99]. These models allow for reasoning about the execution of one software component in isolation. They focus on the semantics of the x86 instruction set and abstract away as many hardware details as possible to increase the applicability of the model. For instance, memories are commonly limited to the DRAM.

This approach works well when software components use a limited amount of hardware features, as applications typically do, and to reason about correctness with respect to functional specifications. It is less suitable in the context of HSE mechanisms verification with respect to security policies, where multiple hardware components are involved, and the many features exposed by the latter may interfere with each other.

It has been shown in the past that it is possible to extend such approaches, when their abstraction is too strong. For instance, Chen *et al.* have extended the hardware model of CompCertX, a variant of CompCert used in the development of formally verified kernels [100], with a notion of input devices and interrupts [101]. However the primarily focus of these

projects is to reason about software components, not the underlying hardware architecture.

Ad-hoc x86 Models Another category of x86 models is ad-hoc models, especially developed to verify a dedicated system software component. As such, they focus on hardware features used by the target of verification. The objective of these approaches is to verify the software component and so they suppose the hardware will behave as expected. On the contrary, we aim to specify requirements on software components, and to use these requirements as assumptions to verify the underlying architecture.

The seL4 microkernel [102] is, to date, the most advanced and mature verified implementation of a microkernel. A implementation in C of the kernel is proven correct with respect to a functional implementation modeled in the Isabelle/HOL theorem prover, and the authors have proven this model correctly enforces that security policies including integrity and confidentiality policies [103]. In practice, the hardware model focuses on MMU, cache and interrupt handling. In large parts the exact behavior of hardware devices is left non-deterministic, in order to reduce the assumptions made by the model for the hardware. Besides, we already mentioned the work of Jomaa *et al.* [11]. In 2016, they have proposed a formal machine-checked proof (in Coq) of guest isolation by an idealized protokernel based on a MMU. Similarly to seL4, their hardware model focus on MMU and interrupt handling. In both cases, there is a clear separation between the software component model and the hardware model, and it may be possible to extract the hardware model and reuse it. However, their scope does not include the hardware features involved in the HSE mechanisms implemented by the BIOS and described in Section 2.2.3.

Between 2011 and 2014, Gilles Barthe *et al.* have worked on an idealized model of a hypervisor [70, 104, 105]. This model is defined in terms of states, actions and the semantics of actions as state transformers. The state definition mixes information about both hardware components (Central Processing Unit (CPU) execution mode, registers, memory content, etc.) and software components (list of guests, the current active guest, memory mapping for the hypervisor and the guests, etc.). The set of actions describes the events which can trigger a transformation of the model states. For instance, it includes various tasks that the hypervisor must carry out, such as scheduling the guests OS, hypercalls handling, or memory management. Certain actions also witness the execution of guests, for instance when the executing guest OS reads from or writes to memory. The resulting project, called VirtCert and implemented in the Coq theorem prover, is fairly large, with over 50,000 lines of code. The verification results focus on various isolation properties, from the most natural and straightforward (*i.e.* an OS guest cannot write to or read from a page it does not own) up to non-interference variants including protection against cache-timing attacks, notoriously harder to reason with. The use of labeled transitions to reason about the interactions of the guests with the underlying system —made of the hardware architecture and the hypervisor— constitutes an interesting approach that we could leverage for modeling the interactions between the software components and the hardware model. However, the focus of VirtCert remains very specific and does not cover the SMM and related mechanisms.

Models for Other Hardware Architectures. The x86 architecture is not the only hardware architecture which has been the object of formal verification projects. We detail two projects targeting ARM and XOM architectures because they follow alternative approaches that are of interest with respect to our goal.

The two latest versions of the ARM processors have been formally specified [106, 107]. It is important to emphasize that the specification of the ARMv8 architectures[1] is the result of an important 5-year effort by ARM Ltd to integrate formal specification definition to their regular specification process [107]. The formal specification of ARMv8 architectures is written in a dedicated language called ARM Specification Language, and has been intensively validated against the ARM internal conformance testsuite.. Once the level of trustworthiness of the ASL specifications have been established, they have been able to leverage them to formally verify properties of the hardware architectures, *e.g.* by compiling them to a subset of Verilog accepted by commercial Verilog model checkers [108]. The result of these efforts is being made available on the ARM website, in addition to regular informal specifications [109]. This represents an exciting opportunity for research targeting ARM architectures, and we can only hope that it eventually becomes a standard in the industry.

ASL is used to *describe* the architecture in an unambiguous fashion, but it cannot be used as-is to reason about the correctness of this architecture. This means the ASL model has to be compiled to another representation, *e.g.* Verilog or a model checker modeling language. On the contrary, we decided to implement our model directly in the language used by the tool supporting our verification process, in order to avoid this translation step.

The eXecute Only Memory (XOM) microprocessor architecture maintains separate so-called *compartments* for applications [110]. A XOM CPU keeps track of each memory location owner, using a tagging mechanism, and supposedly prevents an application from accessing a memory location it does not own. In 2003, David Lie *et al.* have verified the XOM architecture [65] using the Murφ model checker [69]. The verification objective of the authors was to prove that the XOM architecture fulfills its promise to be tamper-resistant, by forbidding an attacker to modify the memory location owned by a given application. The verification proceeds as follows:

1. A first specification of the XOM architecture, called the "actual model", is defined. States of this first model contain different hardware components of a XOM microprocessor, *i.e.* registers, cache, volatile memory, and the internal machinery of XOM to track ownership of memory locations. Transitions can be divided into two categories: the normal execution of an application by the microprocessor, and active tampering from an adversary part, leading the actual model to embed an adversary model.

2. A second specification, called the "idealized model", abstracts away the memory hierarchy formed by the cache and the volatile memory, and models the execution of a single application, without an adversary. From this perspective, it encodes the security property.

[1] There are three variants of the ARM Architecture, for as many use cases: the A-class architecture provides the necessary features to allow an operating system to manage applications, the R-class processors are dedicated to real-time systems, and the M-class are used in microcontrollers.

3. To let Murφ explore both models simultaneously, the authors have manually defined a third model. Transitions which describe the execution of an application in the actual model also update the idealized model, whereas transitions which describe actions by the attacker only affect the actual model.

4. The authors have defined a function which checks if an "actual state" is equivalent to an "idealized state", and let Murφ verify that the state equivalence is an invariant of the third model.

In the process of verifying XOM, the authors have been able to show with their model that the XOM architecture was subject to several replay attacks, and that the countermeasures they proposed were effective. The formalism used to define the actual model follows a logic which is approaching our needs. The labels of the transitions characterize the execution of a software component by the architecture, without any detail in the model about its behavior or its implementation. However, we believe the necessity to manually maintain a merge of two transition systems reduces its applicability in the long run, as new architecture versions are released frequently. This is why our formal theory of HSE mechanism is characterized by a subset of traces of a model, rather than an additional, idealized model.

In order to experiment with our theory, we have chosen to specify and verify the HSE mechanism implemented by the BIOS to remain isolated from the rest of the software stack at runtime. This case study illustrates the challenges faced by the HSE mechanisms in general. To the best of our knowledge, the hardware features involved in this HSE mechanism are outside of the scope of existing models, which is why we developed our own. We drew on modeling approaches like VirtCert [70] and XOM [65], and how they handle the interaction with the rest of the system.

Our theory is characterized by defining a model of the hardware architecture, specifying restrictions over the model, and verifying the resulting restricted model. This methodology is easily expressed in a higher logic. Hardware verification by means of theorem provers has proven to be a practical alternative to model checking [9, 10]. For these reasons, we decided to implement our model and proofs using the Coq theorem prover, to challenge the applicability of our theory with a focused verification problem. This experience convinced us of the interests of a general-purpose hardware model to verify different HSE mechanisms against the same reference, and we have identified several interesting properties for such a model to remain applicable in a theorem prover such as Coq.

3.2 Compositional Verification

The formal verification of a system as complex as the x86 hardware architecture poses significant challenges. This is especially true considering that we focus on security policies and compositional attacks —each hardware feature which is absent from the model can potentially be part

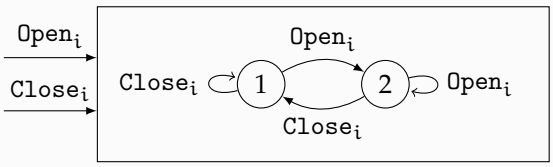
of an attack path. Previous works have advocated for compositional reasoning approaches, such as assume-guarantee [111] or rely-guarantee [112] paradigms, as an interesting solution to tackle these challenges [13, 14]. In the context of compositional reasoning, the system is broken down into several components. A component C is proven to enforce a guarantee G as long as an assumption A is met. If a component C′ is proven to enforce A, then the composition of C and C′ enforces G. Our objective is to apply compositional reasoning paradigms in Coq.

The rest of this Section proceeds as follows. First, we discuss how labeled transition systems can be used to model interacting components (3.2.1), and how process algebra systems have been used to facilitate the definition of these interactions (3.2.2). Finally, we detail how compositional reasoning has been implemented specifically in theorem provers (3.2.3).

3.2.1 Labeled Transition Systems and Components Composition

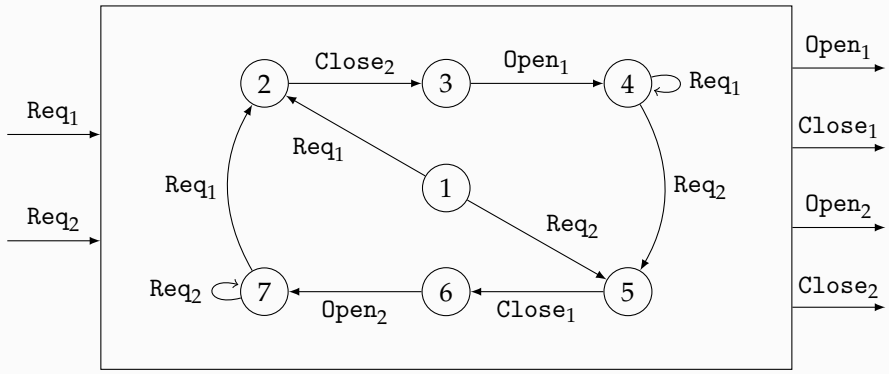
Labeled transition systems have originally been introduced to reason about automata compositions [66], with the idea that transitions of different transition systems which share the same label happen simultaneously. In our context, an interesting variant of LTS is interface automata [113], because they distinguish between three classes of transitions, modeled with three disjoint sets of labels: input actions (denoted by $\mathrm{in}(S)$ for an automaton S), output actions (denoted by $\mathrm{out}(S)$), and internal actions (denoted by $\mathrm{int}(S)$). They form the signature of a given automaton (denoted by $\mathrm{act}(S)$). Its transition relation $R(S)$ is a subset of $\mathrm{state}(S) \times \mathrm{act}(S) \times \mathrm{state}(S)$, where $\mathrm{state}(S)$ is the set of states of S. Composition of interface automata is achieved *via* input and output actions. More precisely, when one automaton performs an output action π during a transition, all automata having π as input action perform π simultaneously.

Example 3.4 (Airlock System as Interface Automata)
In order to illustrate how a system can be broken down into small components, we take once again the example of the airlock system. In this context, the most obvious component is the door. A door has two states: it can either be open or closed. It takes two input actions: `Open` (the action to open the door) and `Close` (the action to close the door). It does not have any output action, which means a door does not interact actively with the rest of the system. One possible specification for a door `i` of an airlock system is the following interface automaton:



In addition to two doors, an airlock system needs a controller, whose purpose is to handle requests coming from users and to effectively open and close doors in consequence. We consider a slightly different situation than the specification given in Example 3.1. Here, there are only two commands, modeled with two input actions: $\mathtt{Req}_1$ (one user wants the first door to be

opened) and $\texttt{Req}_2$ (one user wants the second door to be opened). The controller does not embed the states of the doors, but has four output actions, two per doors ($\texttt{Open}_i$ and $\texttt{Close}_i$, for $i \in \{1, 2\}$). We propose the following interface automaton:



Input and output actions models two facets of a component: how it is being used (input), and how it uses other components (output). However, it is a fairly low-level modeling structure, which complicates its usage for a large-scale model — for instance, our airlock system controller is modeled with seven different states— and impair the model readability. In particular, it is hard to formally establish the causal relation between input and output actions. Rather than defining an automaton manually, it is possible to leverage process algebra formal languages. They allow for modeling, by means of formal languages, interacting components as concurrent processes which exchange messages through channels.

3.2.2 Process Algebra

Process algebras and their proof systems have been developed to reason about programs executed in parallel. In process algebra such as Calculus of Communication Systems [114] or Communicating Sequential Processes [115], concurrent threads run in parallel, and synchronization is achieved by sending and waiting for messages, as specified by a dedicated language.

Using these languages, we can write our "programs of output actions", as we demonstrate in the following example.

Example 3.5 (Airlock System in π-calculus)
We now try to give a specification of our airlock system using a process algebra called π-calculus. Once again, we consider three components: two doors and a controller. Our objective is to write a specification equivalent to our interface automata (although we do not provide a proof of that equivalence).

We have used the following π-calculus construction to specify the airlock system:

– $c(x).P$ means receiving a value from the channel c, bounding this value to the fresh name x,

then executing the process P.

– $\bar{c}\langle x\rangle.P$ means sending the name x through the channel c, then executing the process P.

– $[x = \text{OPEN}]P$ is a guard, that is P is executed if x is equal to the name OPEN.

– $P + Q$ is the nondeterministic choice operator, we use it here in conjunction with guards to implement an `if-then-else` construct. That is, considering the process

$$c(x).([x = 1]P + [x = 2]Q)$$

If the value received from c is 1, P is executed. If it is 2, then Q is executed.

– $\nu c.P$ means a new name c is created, and available for P to use it.

– $P \| Q$ is the parallel execution of P and Q.

$$\begin{aligned}
\text{CloseDoor}(c) &\triangleq c(x).\ ([x = \text{OPEN}]\text{OpenDoor}(c) \\
&\qquad + [x = \text{CLOSE}]\text{CloseDoor}(c)) \\
\text{OpenDoor}(c) &\triangleq c(x).\ ([x = \text{CLOSE}]\text{CloseDoor}(c) \\
&\qquad + [x = \text{OPEN}]\text{OpenDoor}(c)) \\
\text{Controller}(c, d_1, d_2) &\triangleq c(x).\ [x = \text{OPEN}_1]\bar{d_2}\langle\text{CLOSE}\rangle.\bar{d_1}\langle\text{OPEN}\rangle.\text{Controller}(c, d_1, d_2) \\
&\qquad + [x = \text{OPEN}_2]\bar{d_1}\langle\text{CLOSE}\rangle.\bar{d_2}\langle\text{OPEN}\rangle.\text{Controller}(c, d_1, d_2) \\
\text{System} &\triangleq \nu c.\nu d_1.\nu d_2.(\text{Controller}(c, d_1, d_2) \\
&\qquad \| \ \text{CloseDoor}(d_1) \\
&\qquad \| \ \text{CloseDoor}(d_2))
\end{aligned}$$

The system, modeled with the process System, creates the channels used by its components to communicate, then starts their concurrent executions. A door is either open or closed, and we model this with two mutually recursive processes CloseDoor and OpenDoor. They take one channel c as an argument, then wait for new inputs coming from c. A controller is a recursive process which takes three channels c, d_1 and d_2 as arguments. It waits for new requests coming from c. When it receives a new request to open the first (resp. second) door, it first closes the second (resp. first) door, using the channel d_2 (resp. d_1). Then, it opens the first (resp. second) door, using the channel d_1 (resp. d_2).

Process algebras such as π-calculus are well-suited formalisms for describing component interactions. In the context of the previous example, we implement a simple pattern: wait for requests, act accordingly, then start again. A similar approach has been proposed by Choi *et al.* in 2017. They have released Kami [10], a framework for Coq to design, verify and extract (in the form of BlueSpec [116] programs) hardware components implementations. In Kami, components are defined as modules M, that is a particular labeled transition system, whose

transitions are of the form

$$p \xrightarrow[a]{i} (v, q)$$

where i is an operation called by another component, p is the state of the component before the operation i, a is a program of actions performed by the component in order to compute the result of i, v is the result of i that is returned to the caller, and q is the modified state of the component. Actions, in this context, are either local manipulation of the component's state or calls of operations handled by other components.

To reason about M with respect to a specification M_S, Kami introduces a refinement relation $\sqsubseteq$. A module M refines a module M_S (*i.e.* M is an implementation of M_S) if any traces of M can also be produced by M_S. A component M can be composed with another component M' to form a larger component $M + M'$, for instance when the operations exposed by M' are used by M. The authors introduced another "modular" refinement property, whose simplest expression could be

$$M \sqsubseteq N \wedge R \sqsubseteq S \Rightarrow M + R \sqsubseteq N + S$$

They proved the correctness of a realistic multiprocessor system with respect to a simple ISA semantics and Lamport's sequential consistency [117] as the memory model. Their proofs consist of a succession of refinement proofs from their implementation to high-level modules which model instructions semantics and processor memory model.

Kami relies on a very interesting approach to model interconnected components. It extends labeled transition systems to associate programs of output actions to transitions labeled with input actions. However, and similarly to previous hardware verification researches, the proofs focus on properties that are transparent to software components and do not require configuration from their part. The approach used by Kami in order to model interconnect components is very promising, but the verification process relies on successive refinements, rather than a compositional reasoning —which remains our objective. We now give an overview of compositional reasoning framework implemented in theorem provers.

3.2.3 Compositional Reasoning for Theorem Provers

Kami demonstrates that it is possible to model components in isolation, with an emphasis on how they interact together. However, modeling the system is only a first step before the verification process. To our surprise, compositional reasoning in presence of interconnected components has been less studied in the theorem proving community, with the notable exception of the B-method [92].

Several approaches [118, 119, 120] have been proposed to implement compositional reasoning for the B-method, a well-established method of software development based on the B formal system. B allows for the definition of functional specifications, called abstract machines, and the refinement of these specifications up to executable code. An abstract machine is characterized by a set of variables, and several operations which update these variables. Preconditions in the form of predicates on the machine variables can be attached to a given operation to specify when the operation can be used. B has already been used in order to reason about systems

made of components which interact *via* interfaces [118, 119, 120]. In particular, Lanoix *et al.* [120] proposed the following reasoning:

- Interfaces and requirements over components which expose these interfaces are defined in the form of an abstract machine, where the operations preconditions models the assumption a user needs to satisfy.
- A component which exposes this interface is expected to be a refinement (a more concrete implementation) of this abstract machine.
- A model of a component which uses an interface can *include* the abstract machine which models this interface and uses its operations, similarly to what aggregation enables in object-orienting programming language.

This approach implements the key concepts of compositional reasoning we are interested in. However, the B-method proof environment is quite different from other theorem provers, such as Coq. In particular, B is similar to imperative languages, when Gallina is a purely functional language.

To the extent of our knowledge, the `Coq.io` framework, developed and released by Guillaume Claret *et al.* [121], represents the closest approach for Coq, but it addresses a different problem —that is reasoning about functional programs in presence of side effects— and its compositional reasoning approach is far more limited. `Coq.io` only considers two components: a program with side effects, and an outer environment (typically the operating system) which carries out these effects. For example, a program which reads the content of a file relies on an operating system to access the file stored in a hard drive and makes this content available in its address space.

Modeling side effects in pure programming language, such as Haskell or Gallina, is usually achieved with monads [122, 23], and `Coq.io` is no exception. Programs with effects in `Coq.io` are defined in a dedicated monad, with side effects (*e.g.* system calls to read from or write to a file) axiomatized as monadic operations of this monad. The proofs rely on scenarios which determine how an environment (*e.g.* an operating system) would react to the program requests. The verification goal is to verify that, under the hypothesis that the environment is correct with respect to a scenario, then a program with effects is also correct with respect to an expected trace of side effects operations it produces. `Coq.io` paves the road towards a compositional approach reasoning in Coq, but remains incomplete in our perspective. A program with effects can be verified, but the verification of the composition of this program with its environment is out of the scope of the project.

Besides, traditional monadic approaches have a reputation not to compose very well [123], despite constructions such as monad transformers [124]. In our case, we explicitly need to handle the composition of several components, *e.g.* if one component is connected to more than just one. Recently, algebraic effects and effect handlers [24] have been proposed to address this limitation, and they have been the object of implementations for several significant functional programming languages (*e.g.* Eff [24], Idris [125], or Haskell [126]).

We advocate that an approach inspired by algebraic effects and effect handlers can be used to implement a compositional approach for Coq similar to what others have already proposed for the B-method, in particular by Lanoix *et al.* [120]. This is because the separation between program with effects and effect handlers mimics the separation between one component which interacts with a second one. We believe they allow for modeling components similarly to what Kami achieves, but constitute at the same time a more familiar formalism for the functional programming community whom Coq developers belong to.

3.3 Conclusion

In this Chapter, we have explained how it is possible to formally verify a system with respect to a security policy by defining a transition system to model its behavior and reasoning about the set of its traces. This general verification approach does not fully apply in the context of HSE mechanisms, because the hardware architectures are not safe by default. Our theory of HSE mechanisms, presented in Chapter 4, takes this into account. To demonstrate how this theory can be leveraged in practice, we take the example of a HSE mechanism implemented by the BIOS at runtime; and because the scope of existing x86 models does not cover the hardware features that we are interested in, we developed our own in the experiments we detailed in Chapter 5. We decided to use Coq, because its higher logic allows us to easily express the specificities of our approach.

A general-purpose x86 hardware model poses an important challenge due to the scale of the hardware architecture. Previous works have advocated for compositional reasoning approaches, such as assume-guarantee [111] or rely-guarantee [112] paradigms, as an interesting solution to tackle these challenges [13, 14]. However, to our surprise, the subject has not been widely explored in Coq. In Chapter 6, we propose a novel formalism which leverages existing functional programming language paradigms to enable compositional reasoning in Coq.

Part II

Specifying and Verifying HSE Mechanisms

4

A Theory of HSE Mechanisms

"The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise."

— Edsger Dijkstra

Our first contribution is a theory of HSE mechanisms which takes into account that (1) hardware architectures often allow for implementing several HSE mechanisms, and (2) hardware features involved in HSE mechanisms are not safe by default, hence the role played by trusted software components to configure them. The rest of this Chapter proceeds as follows. First, we detail how our theory allows for specifying and verifying HSE mechanisms against general-purpose hardware models, in isolation and in composition (Section 4.1). To demonstrate how we can reason about composition of HSE mechanisms, we proceed through a case study focused on a so-called code injection policy. More precisely, we prove how we can enforce this policy by means of two HSE mechanisms implemented at the same time (Section 4.2). This case study is used as the basis for a more in-depth experiment in the next Chapter.

For readers familiar with Coq, we present an implementation of our theory in Appendix A. This development comprises machine-checked proofs of the theorems and lemmas that we present throughout this Chapter.

4.1 Theory Definition

We model the hardware architecture as a labeled transition system whose sequences of transitions, also called traces, characterize every possible execution of software components (Section 4.1.1). This model acts as a foundation for our HSE mechanisms theory (Section 4.1.2). We then define correctness in the context of HSE mechanisms (Section 4.1.3). Our motivation to separate the model of the hardware architecture and HSE mechanism definitions is to enable

the reuse of the same hardware model to verify the correctness of several HSE mechanisms. Besides, it makes reasoning about HSE mechanisms composition —that is, the implementation, at the same time, of two HSE mechanisms or more— possible (4.1.4).

4.1.1 Hardware Model

We model a hardware architecture, which can execute different software components, as a particular Labelled Transition System (LTS) (Definition 3.1) [1] with two sets of labels instead of one.

States. The set of states of the LTS models possible configurations of the hardware components. This configuration changes over time with respect to the hardware specifications and comprises any relevant data such as register values, inner memory contents, etc. These state transformations occur during the system's transitions.

Labeled Transitions. We distinguish between two classes of transitions: the software transitions which are direct and foreseeable side effects of the execution of instructions and the hardware transitions which are not. Following the terminology of David Basin *et al.* [73], software transitions are "controllable", while hardware transitions are only "observable".

We illustrate this definition with the x86 instruction `mov (%ecx),%eax` [2]. With respect to the semantics of this instruction, a x86 core: (1) reads the content of the register `ecx`, (2) interprets this value as an address and reads the main memory at this address, (3) writes this content into the register `eax`, (4) updates the register `eip` with the address of the next instruction to execute. The execution of this instruction by a core may result in several sequences of transitions. First, the four execution steps described above translate into four software transitions. Between each of these transitions, a hardware transition can occur. For instance, one hardware component can initiate a DMA. Also, if the content of `ecx` is not a valid address, the processor raises an interrupt. In this scenario, only one software transition occurs —when the core reads the content of `ecx`— then the next transition models the interrupt.

Definition 4.1 (Hardware Model)
A hardware model Σ is a tuple $\langle H, L_S, L_H, \rightarrow \rangle$, where

- H is the set of configurations of the hardware architecture
- L_S is the set of labels to identify software transitions
- L_H is the set of labels to identify hardware transitions
- $\rightarrow$ is the transition relation of the system

The transition relation $\rightarrow$ is a predicate on $H \times L \times H$, where $L = L_S \uplus L_H$ and $\uplus$ is the

[1] On page 28.
[2] Written in AT&T syntax here.

disjoint union which requires that $L_S \cap L_H = \emptyset$. A transition labeled with l from h to h' is denoted by

$$h \xrightarrow{l} h'$$

and we write $\mathcal{T}(\Sigma)$ for the set of triples which satisfy the transition relation, that is

$$\mathcal{T}(\Sigma) \triangleq \{ (h, l, h') \mid h \xrightarrow{l} h' \}$$

Traces. A trace is a non-empty sequence of transitions of Σ, such that for two consecutive transitions, the resulting state of the first one is the initial state of the next one.

Definition 4.2 (Traces)
We write $\mathcal{R}(\Sigma)$ for the set of traces of a hardware model Σ, and we consider the following functions:

– $init : \mathcal{R}(\Sigma) \to H$ maps a trace to its initial state

– $trans : \mathcal{R}(\Sigma) \to \wp(\mathcal{T}(\Sigma))$ maps a trace to the set of transitions which occurred during the trace

Mainstream hardware architectures are not "safe" by default, and require additional software configuration to enforce security properties. For instance, it is legitimate, from a x86 specification perspective, to enter ring 3 mode with a page table which allows for modifying kernel code and data. We aim to model HSE mechanisms as subsets of traces, to discard traces that are legitimate from a hardware specification perspective, yet dangerous from a security perspective.

4.1.2 HSE Mechanisms

In our theory, a HSE mechanism is primarily characterized by a set of trusted software components, a set of requirements over states and a set of requirements over software transitions. Trusted software components are responsible for implementing the HSE mechanism. To that end, they have to correctly configure the underlying hardware mechanism in accordance with the two requirements over states and over software transitions.

On the one hand, the purpose of the requirements over states is to identify the hardware configurations which constrain the execution of untrusted software components with respect to a targeted security policy.

On the other hand, the purpose of the requirements over software transitions is to guarantee requirements over states hold true at all time. Because we consider an adversary model where attackers control untrusted software components, we do not consider any hypothesis on their behavior. Therefore, requirements over software transitions should only constrain the execution of trusted software components, which implement the HSE mechanism.

These two classes of requirements allow us to determine a subset of compliant traces, that is traces where trusted software components have correctly implemented a HSE mechanism by satisfying the requirements at all time. From this perspective, to verify whether this HSE

mechanism is correct with respect to a targeted security policy means proving its set of compliant traces satisfies the predicate which models the policy.

HSE Mechanisms. In the context of HSE mechanisms, trusted and untrusted software components alike use the same interface to interact with the underlying hardware components: the processor instructions set. To reason about HSE mechanisms, it is necessary to be able to determine which software component is executed at a given time. In practice, a subset of the states of the hardware architecture is dedicated to each software component. For instance, x86 processors protection rings are commonly used to execute several software components. Ring 0 is dedicated to the operating system, whereas the applications are executed by a core in ring 3, with a particular page table setup.

Definition 4.3 (Hardware-Software Mapping)
Given S a set of software components, a hardware-software mapping $context : H \rightarrow S$ is a function which takes a hardware state and returns the software component currently executed.

This definition is of key importance, because it implies strong hypotheses on how a software stack is executed. First, we assume only one software component is executed at a given time by the hardware architecture, in order to simplify our definitions. We believe dealing with multi-core architectures is possible with the cost of additional efforts —*e.g.* by defining a set of identifiers to select a particular core— yet it remains to be proven as we have not tackled this challenge during this thesis. Besides, there are approaches which allow for executing two software components using the same "hardware context", *e.g.* JavaScript programs of two tabs inside a browser are arguably two distinct software components. However, from the hardware architecture perspective, these three components —the browser and the two JavaScript programs— are indistinguishable, and enforcing a security policy in such a context is achieved by means of software measures, *e.g.* program interpretation or software fault isolation [97].

We say a software transition is trusted (respectively untrusted) when it occurs from a state associated to a trusted (respectively untrusted) software component. A hardware-software mapping is mandatory to define requirements over software transitions that are consistent with respect to our adversary model, *i.e.* which only constrain trusted software components.

Definition 4.4 (HSE Mechanism)
A HSE mechanism Δ is a tuple $\langle S, T, context, hardware\_req, software\_req \rangle$, such that

– S is the set of software components executed by the hardware architecture.

– $T \subseteq S$ is the set of trusted software components which implement the HSE mechanism and form its Trusted Computing Base (TCB).

– *context* is a hardware-software mapping to determine which software component is currently executed by the core.

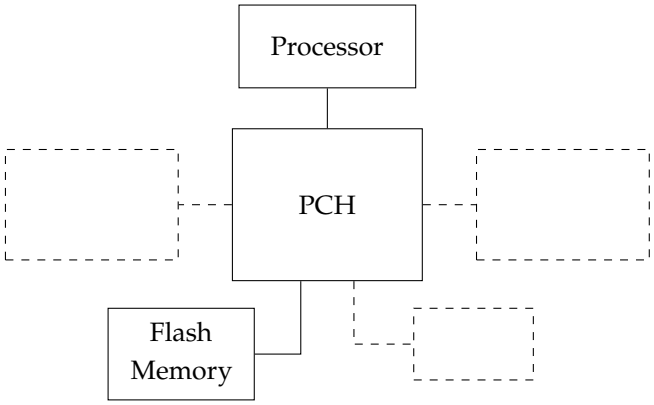


Figure 4.1: From the processor to the flash memory

– *hardware_req* is a predicate on H to distinguish between safe hardware configurations and potentially vulnerable ones.

– *software_req* is a predicate on $\mathsf{H} \times \mathsf{L}_S$ to distinguish between software transitions that trusted software components can safely use, and potentially harmful ones they need to avoid.

We illustrate this definition with the flash memory lockdown mechanism described in Section 2.2.3.

Example 4.1 (Model of Flash Memory Lockdown Mechanism)
As a reminder, such mechanism enforces the integrity of the BIOS code within the flash memory and is implemented by the PCH. As pictured in Figure 4.1, the PCH acts as a proxy between the core and a collection of peripherals, including the flash memory. The HSE mechanism proceeds as follows:

1. By default, the flash memory is locked and its content cannot be modified. By setting the correct bit of a dedicated register of the PCH (`BIOS_CNTL`), a software component requests to unlock the flash memory.

2. When a software component (*e.g.* system software) unlocks the flash memory, the PCH triggers a SMI.

3. This forces the core to enter SMM. By doing so, the core stops the execution of the software component which has unlocked the flash memory and starts the execution of the BIOS.

4. In order to protect the integrity of the flash memory content, the BIOS is expected to lock the flash memory again, before resuming the execution of the system software with the `rsm` assembly instruction.

Legacy BIOSes probably used this mechanism in order to implement an ad-hoc communication channel between the BIOS and the system software component, *e.g.* to initiate a BIOS

update. The UEFI standard introduces dedicated protocols to enable communication between the BIOS and the system software component, and our understanding is that in most implementations the BIOS action, in response to a SMI triggered by the unlocking of the flash memory, is limited to lock it again and resume.

The set of trusted software components is limited to the BIOS, that is $\mathsf{T} = \{\texttt{bios}\}$. This means the rest of the software stack (*e.g.* the operating system and its applications) are untrusted and therefore assumed to be under the control of attackers. The hardware-software *context* maps CPUs in SMM with the execution of the BIOS, that is

$$\begin{array}{lcll} \mathit{context}(\mathtt{h}) & = & \texttt{bios} & \text{if } \mathtt{h} \text{ characterizes a CPU in SMM} \\ \mathit{context}(\mathtt{h}) & \in & \mathsf{S}\backslash\{\texttt{bios}\} & \text{otherwise} \end{array}$$

A safe hardware state (*hardware_req*) is either a state wherein the BIOS is executed, or a state wherein the flash memory is locked. It is expected that the BIOS locks the flash memory before using the `rsm` instruction (*software_req*).

HSE Laws. As it is, our theory of HSE mechanisms is too permissive. For a HSE mechanism to be consistent, it must also obey two requirements, together called the HSE laws. The first law says that the *software_req* predicate always holds true for untrusted software transitions. That is, the first law enforces that a HSE mechanism definition does not make any assumption regarding untrusted software components. The second law says that the requirements over states specified by *hardware_req* are invariant with respect to *software_req*. As long as the trusted software components which implement the HSE mechanism only generate software transitions which satisfy the *software_req* predicate, the *hardware_req* predicate holds true. This means the software transitions at the untrusted software components disposal cannot put the system into an unsafe state.

Definition 4.5 (HSE Laws)
A HSE mechanism Δ has to satisfy the following properties:

1. Untrusted software transitions satisfy *software_req*: $\forall(\mathtt{h}, \mathtt{l}, \mathtt{h}') \in \mathcal{T}(\Sigma)$, $\forall \mathtt{x} \notin \mathsf{T}$,

$$(\mathtt{l} \in \mathsf{L_S} \wedge \mathit{context}(\mathtt{h}) = \mathtt{x}) \Rightarrow \mathit{software\_req}(\mathtt{h}, \mathtt{l})$$

2. *hardware_req* is an invariant with respect to *software_req*: $\forall(\mathtt{h}, \mathtt{l}, \mathtt{h}') \in \mathcal{T}(\Sigma)$,

$$(\mathit{hardware\_req}(\mathtt{h}) \wedge (\mathtt{l} \in \mathsf{L_S} \Rightarrow \mathit{software\_req}(\mathtt{h}, \mathtt{l}))) \Rightarrow \mathit{hardware\_req}(\mathtt{h}')$$

On the one hand, a HSE mechanism definition which does not satisfy the first law is incorrectly making assumptions about "untrusted" software components. This makes these components part of the TCB *de facto*, even though they do not belong to $\mathcal{T}$. Formally defining a HSE mechanism may be the occasion to uncover such implicit assumptions. On the other hand, a HSE mechanism which does not satisfy the second law has to be carefully reviewed. As it

stands, it lets the hardware architecture reach a state wherein the hardware configuration does not satisfy the requirements over states, despite the correct implementation of the mechanism by the trusted software components. In such a condition, the hardware architecture may stop constraining the execution of untrusted software components with respect to the targeted security policy. This could mean that constraints over trusted software components execution are incomplete, or that requirements over hardware states are too restrictive. If the requirements over hardware states cannot be loosened without threatening the security policy enforcement or finding restrictions over trusted software execution, then the HSE mechanism probably does not serve its purpose.

Example 4.2 (Flash Memory Lockdown Inconsistency)
We can convince ourselves that the informal definition description we have discussed in the Example 4.1 obeys the first HSE law, but *does not* obey the second one.

Untrusted software transitions satisfy *software_req***.** The only software restriction we have formulated concerns the use of the `rsm` instruction by the BIOS. The rest of the software stack can leverage the complete x86 instructions set with no restriction. In particular, an attacker is free to unlock the flash memory thanks to the `BIOS_CNTL` register.

hardware_req **is not an invariant with respect to** *software_req***.** A CPU is either in SMM or not in SMM. If it is in SMM, the only way to leave SMM is to execute the `rsm` instruction, which qualifies as a software transition. If the CPU is in SMM, the *software_req* hold true for this software transition when the flash memory is locked. As a consequence, a BIOS which correctly implements this HSE mechanism locks the flash memory prior to leaving the SMM. However, if the CPU is not in SMM, and tries to open the flash memory, two things happen sequentially. First, the flash memory is effectively opened. This has to be modeled by a software transition. Then, the PCH triggers a SMI, leading the CPU to enter SMM. A SMI has to be modeled as a hardware transition. Between the modification of the `BIOS_CNTL` register and the treatment of the SMI by the CPU, the *hardware_req* predicate is not satisfied.

It is because the flash memory lockdown as described here does not satisfy the first law that the race condition uncovered by Corey Kallenberg *et al.* [7] is possible. On the contrary, the `SMM_BWP` register semantics —detailed in Subsection 2.3.2— allows for defining a HSE mechanism which satisfies both laws, because the register ties together the state of the CPUs and the flash memory state.

Trace Compliance. Whether trusted software components are correctly implementing a given HSE mechanism is a safety property: trusted software components shall *never* generate a software transition which does not satisfy the requirements of the HSE mechanism. The purpose of these requirements over software transitions is to prevent untrusted software components to reach a hardware configuration wherein the requirements over states are not satisfied. This implies that the initial state of the trace must satisfy the requirements over states. A trace whose

initial state satisfies the requirements over states and where trusted software components do correctly implement a HSE mechanism is said to comply with this mechanism.

Definition 4.6 (Compliant Traces)
We write $\mathcal{C}(\Delta)$ for the set of the traces which comply with Δ. Given $\rho \in \mathcal{R}(\Sigma)$, then $\rho \in \mathcal{C}(\Delta)$ iff

$$\mathit{hardware\_req}(\mathit{init}(\rho)) \wedge \forall (h, l, h') \in \mathit{trans}(\rho),\ l \in L_S \Rightarrow \mathit{software\_req}(h, l)$$

Example 4.3 (Memory Flash Lockdown Compliance)
We consider the trace to start at the end of the boot sequence, when the BIOS gives the control flow to the system software component it has selected. At the time, it is expected that the `BIOS_CNTL` register of the PCH has been correctly set; in particular, the `SMM_BWP` bit should be set, to avoid the Speed Racer attack [7]. If the BIOS fails to do so, then the related trace is not compliant to begin with. If it succeeds, then it still needs to carefully satisfy its requirements over transitions throughout its execution for the trace to remain compliant over time.

Lemma 4.1 (HSE Invariant Enforcement)
As intended, *hardware_req* is an invariant of traces which comply with Δ, that is

$$\forall \rho \in \mathcal{C}(\Delta), \forall (h, l, h') \in \mathit{trans}(\rho), \mathit{hardware\_req}(h) \wedge \mathit{hardware\_req}(h')$$

Proof. By definition of $\mathcal{C}(\Delta)$, we know the initial state of the trace satisfies *hardware_req*, and thanks to the second HSE law, we can conclude the state after the first transition also satisfies *hardware_req*. We generalize to the trace by induction. □

As a consequence, a HSE mechanism allows for satisfying a given set of requirements over hardware configurations throughout the execution of a software stack, as long as a set of trusted software components correctly implement the mechanism.

4.1.3 HSE Mechanism Correctness

It is important to keep in mind that preserving a set of requirements over states is only a means to an end. The true purpose of a HSE mechanism is to constrain the execution of untrusted software components with respect to a targeted security policy.

A HSE mechanism can satisfy the HSE laws, but yet fails to constrain the execution of untrusted software components with respect to a targeted security policy. The SMRAM cache poisoning attack disclosed in 2009 is a good illustration of that eventuality [4, 5]. Back then, the isolation of the BIOS at runtime was enforced by only two hardware mechanisms: the SMM of the CPU and the `SMRAMC` register of the memory controller. By correctly configuring the memory controller *via* the `SMRAMC` register, the BIOS can activate the protection of a dedicated memory

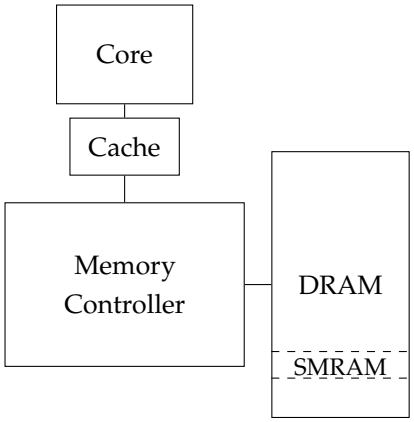


Figure 4.2: From the core to the SMRAM

region called the SMRAM. In such a case, only a core in SMM can read or write to the SMRAM. As it stands, the HSE mechanism described here satisfies both laws. The rest of the software component can try to read from or write to the SMRAM (first law), but the memory controller will protect its integrity in the presence of a core not in SMM (second law). However, between the core and the memory controller lies the cache, as pictured in Figure 4.2. If the cache does not implement an access control mechanism to protect its copies of the SMRAM content —as was the case in 2009— it can be used to circumvent the protection enforced by the memory controller. As we have already explained in Subsection 2.3.1, the integrity of the SMRAM is enforced, so only a core in SMM can access to the content of the SMRAM. However, when this core leaves the SMM, the content of the SMRAM remains in the cache, and nothing prevents an untrusted software component to modify these copies. When the core enters in SMM again, it fetches the (modified) content of the cache in place of the true SMRAM content.

We detailed in Subsection 3.1.2 how various classes of security policies —safety and liveness properties, and hyperproperties— can be modeled against a transition system. Because our hardware model is a labeled transition system, we can easily transpose these definitions to our theory.

Definition 4.7 (Security Policy)
A security policy $\mathtt{P}$ is either a predicate on sets of traces, a predicate on traces or a predicate on transitions.

Hardware features leveraged in the context of a HSE mechanism Δ are not safe by default and require to be configured by trusted software components. As a consequence, the security policy $\mathtt{P}$ targeted by Δ does not apply to the set of all traces of the hardware model, but we expect that satisfying the requirements of Δ —which is modeled in our theory by the set of compliant traces $\mathcal{C}(\Delta)$— is a sufficient condition for enforcing $\mathtt{P}$.

Definition 4.8 (Correct HSE Mechanism)
A HSE mechanism Δ is correct with respect to a security policy P (denoted by $\Delta \models \mathsf{P}$) if and only if the set of compliant traces of Δ satisfies P, that is

$$\Delta \models \mathsf{P} \triangleq \begin{cases} \mathsf{P}(\mathcal{C}(\Delta)) & \text{if } \mathsf{P} \text{ is a predicate on sets of traces} \\ \forall \rho \in \mathcal{C}(\Delta), \mathsf{P}(\rho) & \text{if } \mathsf{P} \text{ is a predicate on traces} \\ \forall \rho \in \mathcal{C}(\Delta), \forall tr \in \mathit{trans}(\rho), \mathsf{P}(tr) & \text{if } \mathsf{P} \text{ is a predicate on transitions} \end{cases}$$

Verifying that a HSE mechanism is correct with respect to a security policy can be difficult. In the context of this thesis, we focus on safety properties. To the extent of our knowledge, the majority of HSE mechanisms target access control policies, which qualify as safety properties. The reasoning is facilitated because they are characterized by predicates on transitions. The following theorem takes advantage of this fact.

Theorem 4.1 (Correct HSE Mechanism for Predicate on Transitions)
Given a security policy P defined as a predicate on transitions, then

$$\begin{aligned} &\forall (\mathsf{h}, \mathsf{l}, \mathsf{h}') \in \mathcal{T}(\Sigma), \\ &\quad (\mathit{hardware\_req}(\mathsf{h}) \wedge (\mathsf{l} \in \mathsf{L_S} \Rightarrow \mathit{software\_req}(\mathsf{h}, \mathsf{l}))) \Rightarrow \mathsf{P}(\mathsf{h}, \mathsf{l}, \mathsf{h}') \end{aligned}$$

is a sufficient condition for

$$\Delta \models \mathsf{P}$$

Proof. By definition, a HSE mechanism Δ is correct with respect to a security policy characterized by a predicate on transitions P if the transitions of its compliant traces satisfy P. With Lemma 4.1, we know that initial states of a transition of compliant traces satisfy *hardware_req*. We also know by definition of compliant traces that their transitions satisfy *software_req*. Therefore, if we can prove that transitions which satisfy both *hardware_req* and *software_req* also satisfy P, then we can conclude that transitions of compliant traces satisfy P. □

A proof of the correctness of a given HSE mechanism asserts that the hardware architecture enforces a targeted security policy *if trusted software components correctly implement this HSE mechanism*, that is their executions only imply software transitions which satisfy the appropriate requirements. If attackers are able to modify the instructions of the trusted software components programs, they can defeat the security enforcement by modifying the instructions responsible for implementing it. This is why trusted software components are expected to protect themselves against untrusted software components attempt to modify their program. They satisfy this expectation thanks to dedicated HSE mechanisms that we can also verify. This raises the question of the implementation of several HSE mechanisms at the same time.

4.1.4 HSE Mechanisms Composition

In common software stacks, several HSE mechanisms are implemented at the same time by low-level software components. The fact that two HSE mechanisms of the same hardware architecture can be specified against the same general-purpose hardware model allows us to reason about their *composition*. We model the concurrent implementation of two HSE mechanisms Δ_1 and Δ_2 as a third HSE mechanism $\Delta_1 \sqcap \Delta_2$. For convenience, the $\sqcap$ operator imposes two restrictions over HSE mechanisms: they shall share the same set of software components and the same hardware-software mapping.

Definition 4.9 (Concurrent HSE Mechanisms)
Given two HSE mechanism Δ_1 and Δ_2, such that

$$\begin{aligned} \Delta_1 &= \langle S, T_1, \mathit{context}, \mathit{hardware\_req}_1, \mathit{software\_req}_1 \rangle \\ \Delta_2 &= \langle S, T_2, \mathit{context}, \mathit{hardware\_req}_2, \mathit{software\_req}_2 \rangle \end{aligned}$$

We write $\Delta_1 \sqcap \Delta_2$ for the HSE mechanism which combines the requirements of both Δ_1 and Δ_2, that is

$$\Delta_1 \sqcap \Delta_2 \triangleq \langle S, T_1 \cup T_2, \mathit{context}, \mathit{hardware\_req}_{1\wedge 2}, \mathit{software\_req}_{1\wedge 2} \rangle$$

where

$$\begin{aligned} \mathit{hardware\_req}_{1\wedge 2}(h) &\triangleq \mathit{hardware\_req}_1(h) \wedge \mathit{hardware\_req}_2(h) \\ \mathit{software\_req}_{1\wedge 2}(h, l) &\triangleq \mathit{software\_req}_1(h, l) \wedge \mathit{software\_req}_2(h, l) \end{aligned}$$

We now discuss two properties of $\sqcap$ which emphasize that its definition matches legitimate expectations. Firstly, $\sqcap$ forms a commutative monoid, and as a consequence is both associative and commutative. This means the order in which we define a "composite" HSE mechanism from a list of sub-mechanisms is not important. Secondly, the set of compliant traces of the composition of two mechanisms is the intersection of the sets of compliant traces of these mechanisms. In other words, to comply with the composition of two HSE mechanisms, a trace has to comply with each mechanism individually.

Lemma 4.2 (Commutative Monoid)
We write $\Delta_\top$ for the HSE mechanism whose requirements over states and software transitions are always satisfied. $\sqcap$ forms a commutative monoid with the set of HSE mechanisms which share both the same set of software components and the same hardware-software mapping, whose identity element is $\Delta_\top$.

Indeed, $\sqcap$ satisfies the following properties:

Associativity: $(\Delta_1 \sqcap \Delta_2) \sqcap \Delta_3 = \Delta_1 \sqcap (\Delta_2 \sqcap \Delta_3)$

Identity Element: $\Delta \sqcap \Delta_\top = \Delta$

Commutativity: $\Delta_1 \sqcap \Delta_2 = \Delta_2 \sqcap \Delta_1$

Proof. $\sqcap$ is defined with operators which themselves form monoids. For each property, the proof consists of unfolding the definition of $\sqcap$ and relying on the properties of $\cup$ and $\wedge$. □

Lemma 4.3 (Compliant Traces, Composition and Intersection)
The set of compliant traces of the intersection of Δ_1 and Δ_2 is the intersection of the sets of compliant traces of Δ_1 and Δ_2, that is

$$\mathcal{C}(\Delta_1 \sqcap \Delta_2) = \mathcal{C}(\Delta_1) \cap \mathcal{C}(\Delta_2)$$

Proof. The main idea of the proof is to leverage the definition of $\wedge$ and $\Rightarrow$ to turn a statement of the form

$$(P \wedge P') \wedge (R \Rightarrow Q \wedge Q')$$

into

$$(P \wedge (R \Rightarrow Q)) \wedge (P' \wedge (R \Rightarrow Q'))$$

and vice versa. In our case, P and P' are statements about the initial states of traces and *hardware_req*, R is the premise filtering software transitions, and Q and Q' are statements about *software_req*. □

The definition of a composition operator poses the question of HSE mechanisms compatibility. There are obvious scenarios where two HSE mechanisms Δ_1 and Δ_2 can doubtlessly be considered incompatible. For instance, $\mathcal{C}(\Delta_1 \sqcap \Delta_2) = \emptyset$ would mean it is not possible to implement both at the same time. We anticipate there are other scenarios more unclear, for instance if the set of compliant traces $\mathcal{C}(\Delta_1 \sqcap \Delta_2)$ contains only traces where the execution of trusted software components are secure, but is not able to serve its purpose. We argue that a similar situation can occur for a single HSE mechanism. Because we aimed to focus on hardware architecture verification in the context of this thesis, we therefore did not investigate further the compatibility of HSE mechanisms.

4.2 Case Study: Code Injection Policy

In the previous Section, we introduced our theory to formally specify and verify HSE mechanisms. It allows us to verify each HSE mechanism in isolation, and we also introduced a composition operator as a first step toward reasoning about the implementation of two HSE mechanisms or more at the same time. We now aim to illustrate further how our theory can be leveraged in order to reason about security enforcement by means of several HSE mechanisms implemented at the same time. To that end, we take as an example a security policy which explicitly forbids software components of a typical software stack —that is, the BIOS, an operating system and several applications— to perform illegitimate code injection (4.2.2). As we discussed in Subsection 4.1.3, the integrity of the trusted software components programs is an implicit hypothesis of our approach, which makes our code injection policy a prerequisite for

reasoning about other HSE mechanisms. We detail how this security policy can be enforced through the correct implementation of two HSE mechanisms (4.2.3).

4.2.1 Defining Code Injection

A memory location within a hardware architecture is a container dedicated to store data used by software components *e.g.* a general-purpose register of a processor, a DRAM memory cell, etc. To formally define a code injection, it is necessary to be able to map an instruction executed by the processor to the software component which has written this instruction into its memory location. To that end, we assign to each memory location a so-called *owner*, such that a software component becomes the new owner of a memory location when it overwrites its content during a software transition. A hardware model tracks the memory location ownership when (1) the hardware architecture state maps each memory location with a software component called its *owner*, and (2) its transition relation updates this mapping throughout traces. By extension, we say a software component owns some data when it owns the memory location in which these data are stored.

Definition 4.10 (Transition-Software Mapping)
A transition-software mapping $\mathit{fetched}_\Sigma : H \times L \to \wp(S)$ is a function which takes an initial hardware state, a transition label and returns the set of owners of the instructions read during this transition by the core in order to execute them afterwards. The word "fetch" is used in the hardware industry to describe this subset of read accesses targeting instructions.

With this mapping, specific to a given hardware model, it becomes possible to determine the owner of an instruction fetched by the processor in order to be decoded and executed. $s \in \mathit{fetched}_\Sigma(h, l)$ means that an instruction owned by s was fetched in order to be executed by the processor during a transition labeled with l from a state h.

With a hardware-software mapping and a transition-software mapping, we give a formal definition of a *code injection*.

Definition 4.11 (Code Injection)
A software component $x \in S$ achieves a code injection against another software component $y \in S$ during a transition labeled with $l \in L$ from a state $h \in H$ to a state $h' \in H$, denoted by

$$h \xrightarrow[x \rightsquigarrow y]{l} h'$$

when the processor fetches an instruction owned by x while executing y, that is

$$h \xrightarrow[x \rightsquigarrow y]{l} h' \triangleq x \in \mathit{fetched}_\Sigma(h, l) \wedge \mathit{context}(h) = y$$

We write $h \xrightarrow[x \not\rightsquigarrow y]{l} h'$ when x does not achieve a code injection against y.

It is important to emphasize that a code injection is not inherently a "bad thing." A software component initialization is most of the time the result of a code injection.

Example 4.4 (Application Code Injection)
When a user executes a new application, the operating system loads the code of this application into the DRAM. Therefore, when the processor starts the execution of the application in ring 3, it executes instructions which are owned by the OS.

On the contrary, an illegitimate code injection poses a significant threat against the security of the software stack, because it constitutes an attack vector for privilege escalation. Low-level software components, such as the BIOS, partly rely on the hardware architecture to protect them against illegitimate code injection. They store their code in a memory region of their choice and implement a HSE mechanism to prevent upper layers of the stack from modifying its content. As a consequence, unsound HSE mechanisms pave the road to illegitimate code injection.

Example 4.5 (SMRAM Cache Poisoning)
The SMRAM cache poisoning attack [4, 5] that we detailed in Subsection 2.3.1 can be used to perform a code injection attack against the BIOS. For instance, Loic Duflot *et al.* have been able to inject the necessary instructions inside the cache for the processor so that the BIOS updates the `SMBASE` register, whose purpose is to determine the first instruction executed by a core when it enters SMM.

4.2.2 Code Injection Policy

We leverage the code injection definition to propose a generic formalization of code injection policy. This policy determines if a given software component is authorized to make a code injection against another software component.

Definition 4.12 (Code Injection Policy)
A code injection policy I is a safety property characterized by a tuple $\langle S, \rightarrowtail, context \rangle$, such that

– S is a set of software components executed by the hardware architecture

– $\rightarrowtail$ is a binary relation on S, such that given $(x, y) \in S \times S$, $x \rightarrowtail y$ means x is authorized to make a code injection against y. $\rightarrowtail$ has to be (1) anti-symmetric, and (2) transitive.

$$\forall (x, y) \in S \times S, \quad x \rightarrowtail y \wedge y \rightarrowtail x \Rightarrow x = y \quad (1)$$

$$\forall (x, y, z) \in S \times S \times S, \quad x \rightarrowtail y \wedge y \rightarrowtail z \Rightarrow x \rightarrowtail z \quad (2)$$

– *context* is a hardware-software mapping.

A transition labeled with $l \in L$ from $h \in H$ to $h' \in H$ satisfies I when code injections which occur during this transition are authorized by $\rightarrowtail$, that is

$$I(h, l, h') \triangleq \forall (x, y) \in S \times S,\ h \xrightarrow[x \rightsquigarrow y]{l} h' \Rightarrow x \rightarrowtail y$$

Based on Definition 4.12, we can define a code injection policy for a typical software stack which comprises the BIOS, an operating system and n applications.

Definition 4.13 (BIOS-OS-Applications Code Injection Policy)
For a software stack made of a BIOS, an operating system and n applications, that is

$$S \triangleq \{\texttt{bios}, \texttt{os}, \texttt{app}_1, \ldots \texttt{app}_n\}$$

we define the authorization relation $\rightarrowtail$ using three rules:

$\rightarrowtail$**-refl:** A software component is authorized to tamper with its own execution, that is

$$\forall x \in S, x \rightarrowtail x$$

$\rightarrowtail$**-bios:** The BIOS is authorized to tamper with the execution of the rest of the software stack, that is

$$\forall x \in S, \texttt{bios} \rightarrowtail x$$

$\rightarrowtail$**-os:** The OS is authorized to tamper with the execution of the application it manages, that is

$$\forall k \in [0, n], \texttt{os} \rightarrowtail \texttt{app}_k$$

We prove by case enumeration that $\rightarrowtail$ is anti-symmetric and transitive.

The definition of the corresponding *context* shall obey the following logic: when the processor is in SMM, the BIOS is executed; when the processor is not in SMM and is in ring 0, the operating system is executed; when the processor is not in SMM and is in ring 3, one of the applications —identified by the page tables used by the processor— is executed.

4.2.3 Code Injection Policy Enforcement

A security policy such as the one detailed in Definition 4.13 is not enforced by one but several HSE mechanisms. Therefore, we break it down to more specific security sub-policies, with each one being enforced by a dedicated HSE mechanism.

Definition 4.14 (BIOS Code Injection Sub-policy)
We write $I_{\texttt{bios}}$ for the code injection sub-policy, whose purpose is to enforce that the only software component authorized to perform a code injection against the BIOS is the BIOS itself,

that is

$$I_{\texttt{bios}}(h, l, h') \triangleq \forall x \in S, h \xrightarrow[x \rightsquigarrow \texttt{bios}]{l} h' \Rightarrow x = \texttt{bios}$$

Definition 4.15 (OS Code Injection Sub-policy)
We write $I_{\texttt{os}}$ for the code injection sub-policy, whose purpose is to enforce that applications are only authorized to perform code injection against themselves, that is

$$I_{\texttt{os}}(h, l, h') \triangleq \forall x \in S, h \xrightarrow[\texttt{app}_k \rightsquigarrow x]{l} h' \Rightarrow x = \texttt{app}_k$$

Without any knowledge about the HSE mechanisms used to enforce $I_{\texttt{bios}}$ and $I_{\texttt{os}}$, we can prove they together imply the enforcement of I.

Theorem 4.2 (Code Injection Policy Enforcement)
Given two HSE mechanisms $\Delta_{\texttt{os}}$ and $\Delta_{\texttt{os}}$, if $\Delta_{\texttt{bios}}$ is correct with respect to $I_{\texttt{bios}}$ and $\Delta_{\texttt{os}}$ is correct with respect to $I_{\texttt{os}}$, then $\Delta_{\texttt{bios}} \sqcap \Delta_{\texttt{os}}$ is correct with respect to I, that is

$$(\Delta_{\texttt{bios}} \models I_{\texttt{bios}} \wedge \Delta_{\texttt{os}} \models I_{\texttt{os}}) \Rightarrow \Delta_{\texttt{bios}} \sqcap \Delta_{\texttt{os}} \models I$$

Proof. By definition of correct HSE mechanism, we need to prove that the set of traces which comply with $\Delta_{\texttt{bios}} \sqcap \Delta_{\texttt{os}}$ satisfies the security policy characterized by I. With Lemma 4.3, we know that the set of compliant traces of $\Delta_{\texttt{bios}} \sqcap \Delta_{\texttt{os}}$ is the intersection of the sets of compliant traces of $\Delta_{\texttt{bios}}$ and $\Delta_{\texttt{os}}$. As a consequence, we know any transition $h \xrightarrow{l} h'$ of these traces satisfy both $I_{\texttt{os}}$ and $I_{\texttt{bios}}$. We can conclude by case enumeration:

– I does not impose any restriction if the `bios` performs a code injection during such a transition.

– Because of $I_{\texttt{bios}}$, an `os` can only perform a case injection against itself are against an application, which satisfies I

– Because of $I_{\texttt{os}}$, an application can only perform a case injection against itself, which also satisfies I. □

In this Section, we specified a general-purpose policy called code injection policy, and detailed how we could apply it to a typical software stack made of a BIOS, an operating system and n applications. More precisely, we proved it is possible to enforce such a policy in this setup by enforcing two sub-policies (respectively $I_{\texttt{bios}}$ to protect the BIOS, and $I_{\texttt{os}}$ to constrain the execution of the applications) thanks to two HSE mechanisms (respectively denoted by $\Delta_{\texttt{bios}}$ and $\Delta_{\texttt{os}}$). On a typical x86 hardware architecture, $\Delta_{\texttt{bios}}$ relies on the SMM and the SMRAMC memory region within DRAM (as we detailed in Subsection 2.2.3), and $\Delta_{\texttt{os}}$ relies on the MMU and the protection rings.

4.3 Conclusion

In this Chapter, we have presented our theory of HSE mechanisms, in the form of requirements over a hardware model. We emphasize that our approach can be used to verify several HSE mechanisms against the same hardware model. We believe this is an essential property; in practice, these multiple mechanisms are implemented at the same time by different software components, and they may interfere with each other, as illustrated by the SENTER Sandman attack discussed in Subsection 2.3.3.

Throughout this Chapter, we have tried to illustrate our definitions with real-world examples, as our effort has been originally motivated by the disclosure of several vulnerabilities targeting multiple x86 HSE mechanisms for the past few years [5, 4, 127, 6, 7]. All being told, our approach can be summarized to a three-step methodology for specifying and verifying HSE mechanisms against hardware architecture models, that is (1) specifying the software requirements that must be satisfied by the trusted software components which implement the HSE mechanism, (2) specifying the targeted security policy the HSE mechanism supposedly enforces, and (3) verifying that the HSE mechanism is correct with respect to the targeted security policy. We believe this methodology would benefit both hardware designers and software developers, but we are interested in challenging the applicability of our approach. To that end, we proceed with the case study introduced in Section 4.2, by specifying the HSE mechanism implemented by the BIOS to remain isolated from the rest of the software stack after the end of the boot sequence (detailed Section 2.3), and verifying this HSE mechanism is correct with respect to the BIOS code injection (Definition 4.14).

5

Specifying and Verifying a BIOS HSE Mechanism

"BIOS as the root of trust. For everything."

— Joanna Rutkowska

In Chapter 4, we have introduced a theory of HSE mechanisms, whose purpose is to be the foundation of a three-step methodology to formally specify and verify HSE mechanisms. We have also introduced a code injection policy for a software stack made of a BIOS, an operating system and n applications (Definition 4.13), and we have defined a sub-policy which focuses on the BIOS protection (4.14). In this Chapter, we apply our approach to specify and verify the HSE mechanism implemented by the BIOS on x86 hardware architectures at runtime and described in Section 2.3. The purpose of this HSE mechanism is to provide an isolated execution environment to the BIOS. It should therefore enforce the so-called BIOS code injection sub-policy.

The rest of this Chapter proceeds as follows. First, we define a minimal x86 hardware model we call Minx86, whose scope comprises the hardware features required to express our case study (Section 5.1). Then, we give a formal definition of the HSE mechanism implemented by the BIOS at runtime, and show that this definition satisfies the two HSE laws and is correct with respect to the targeted security policy previously introduced (Section 5.2). The model and the proofs described in this Chapter have been implemented using Coq. We have released this development as free software in a project called SpecCert [1].

5.1 A Minimal x86 Hardware Model

Minx86 is intended to be a minimal model for single core x86-based machines and we have used publicly available Intel documents [128, 35, 34] to define it. The hardware architecture

[1] SpecCert can be found at the following URL: https://github.com/lthms/speccert

we are modeling with Minx86 matches the one depicted in Figure 4.2. It contains one core[2], one level of cache, a memory controller, a DRAM controller and a VGA controller[3] which both expose some memory to the core. In its current state of implementation, its scope focuses on the SMM, but Minx86 is intended to be incrementally extendable to cover more hardware features.

The rest of this Section proceeds as follows. We give an overview of the Minx86 scope (5.1.1), then describe in depth its key components of the LTS, that is its set of states (5.1.2) on the one hand, and its set of labeled transitions (5.1.3) on the other hand. Finally, in order to reason about code injection policies (Definition 4.12), we define the transition-software mapping of Minx86 (5.1.4).

5.1.1 Model Scope

Hardware Features. We consider the core can be either in SMM or in an unprivileged mode. As explained in Section 2.2.3, when a processor receives a System Management Interrupt (SMI), it halts its current execution, reconfigures itself to a hard-coded state, and then executes the code stored in memory at the address `SMBASE` + `0x8000`. In practice, the SMBASE value points to the beginning of a memory region called the SMRAM. Leaving the SMM is done by executing a special purpose instruction called `rsm` (for *resume*).

The core relies on a cache to reduce the I/O latency. We model one level of cache which stores both data and instructions and we consider two cache strategies: uncacheable (UC) and writeback (WB). With the UC strategy, the cache is not used and all I/Os are forwarded to the memory controller, whereas with the WB strategy, the cache is used as much as possible[4]. To determine which cache strategy to use, the core relies on several configuration registers. Among them, the System Manangement Range Registers (SMRR) tell the core where the SMRAM is located and which cache strategy to use for I/O targeting the SMRAM when the core is in SMM. When it is not in SMM, the core always uses the UC strategy for I/O targeting the SMRAM. Such registers can only be configured when the core is in SMM. SMRR have been introduced as a countermeasure to the SMRAM cache poisoning attack [5, 4] which allowed some untrusted code to tamper with the copy of the SMRAM stored in the cache.

The memory controller [35] receives all the core I/Os which are not handled by the cache and dispatches them to the DRAM controller or to the VGA controller. It exposes a unified view (the memory map) of the system memory to the core. The core manipulates this memory map with a set of physical addresses and the memory controller uses a special range of these physical addresses to host the SMRAM. This memory area is dedicated to store the code intended to be executed when the core is in SMM.

Tracking Memory Ownership. The Minx86 definition is parameterized by a hardware-software mapping *context* (Definition 4.3). The memory locations of Minx86 are either cache

[2] As discussed in the previous Chapter, we did not tackle the challenge posed by multi-core computers in this thesis.

[3] A VGA controller is a hardware device on which we can connect a screen. It exposes some memory to the core for communication purposes.

[4] These cache strategies are explained in [34], Volume 3A, Chapter 11, Section 11.3 (page 2316 – 2317)

lines or memory cells exposed by the DRAM or the VGA controllers. The memory ownership is updated through transitions according to three rules:

1. When a cache line gets a copy of a DRAM or VGA cell content, the owner of this cell becomes the new owner of this cache line.

2. When the content of a cache line is written back to a memory cell, the new owner of this memory cell is the owner of the cache line.

3. During a software transition whose purpose is to overwrite the content of a memory location, the software component currently executed —and therefore responsible for the software transition— becomes the new owner of this memory location.

Definition 5.1 (Minx86)
We write MINX86(*context*) for the hardware model which tracks memory ownership with respect to the *context* hardware-software mapping, such that

$$\text{MINX86}(\mathit{context}) \triangleq \langle \mathrm{H}(\mathrm{S}), \mathrm{L}_\mathrm{S}, \mathrm{L}_\mathrm{H}, \xrightarrow[\mathit{context}]{} \rangle$$

where S is the image of *context*, that is the set of software components which form the software stack.

We emphasize that there are many valid definitions of the *context* function, regarding the verification problem we consider. For instance, we need two different definitions of *context* whether we consider a hypervisor in the software stack or not. We do not rely on the concrete definition of *context* to implement the tracking of memory ownership within MINX86, hence it is left as a parameter for the model.

5.1.2 Hardware States

H(S) is defined as the Cartesian product of the set of states of the core, the cache, the memory controller and the memories exposed by both the DRAM and the VGA controllers. Each of these sets is defined in order to model the hardware features we have previously described. We use named tuples [5] to define them. Thus, H(S) is defined as follows:

$$\mathrm{H}(\mathrm{S}) \triangleq \langle\ \mathit{core} : \texttt{Core},\quad \mathit{cache} : \texttt{Cache}(\mathrm{S}),\quad \mathit{mc} : \texttt{MC},\quad \mathit{mem} : \texttt{Mem}(\mathrm{S})\ \rangle$$

`Cache` and `Mem` are parameterized by S because they expose memory locations to the core which can contain instructions to be fetched. To implement the tracking of memory ownership within our model, we need to taint the memory locations with their respective owner, which belongs to S.

[5] See page xi for a description of the notations we used to manipulate them.

Address Spaces. We define PhysAddr, the set of physical addresses that the core uses to perform I/O and HardAddr, the set of hardware addresses exposed by the DRAM and VGA controllers. To distinguish between DRAM and VGA addresses, we use two different constructors [6].

$$\begin{array}{rcl} \texttt{PhysAddr} & \triangleq & \textsc{Pa} : [0, \texttt{max\_addr}] \to \texttt{PhysAddr} \\ \\ \texttt{HardAddr} & \triangleq & \textsc{Dram} : [0, \texttt{max\_addr}] \to \texttt{HardAddr} \\ & | & \textsc{Vga} : [0, \texttt{max\_addr}] \to \texttt{HardAddr} \end{array}$$

The maximal address offset (denoted by max_addr here) is specific to the core and may vary in time according to its addressing mode (real mode, long mode, etc.), therefore we leave its value as a parameter of our model. By convenience, we give the same maximum address to each address space.

Finally, we write Val for the set of values that the memory cells scattered within the memory locations of the hardware architecture can take.

Core. The set of states of the core is denoted by Core. We give a minimal definition of Core, with a clear focus on the SMM.

$$\texttt{Core} \triangleq \left\langle \begin{array}{c} \mathit{in\_smm} : \{\texttt{true}, \texttt{false}\}, \quad \mathit{pc} : \texttt{PhysAddr}, \quad \mathit{smbase} : \texttt{PhysAddr}, \\ \mathit{smrr} : \texttt{Smrr}, \quad \mathit{strat} : \texttt{PhysAddr} \to \texttt{CacheStrat} \end{array} \right\rangle$$

The boolean *in_smm* is set to true when the core is in SMM and to false otherwise. The physical address *pc* models the program counter, a register used to store the address of the next instruction to be fetched and executed. The physical address *smbase* models the register of the same name. The map *strat* abstracts away the numerous mechanisms of x86 microprocessors to determine which cache strategy to use for a given I/O, where $\texttt{CacheStrat} \triangleq \{\texttt{UC}, \texttt{WB}\}$ is the set of the modeled cache strategies. The set of states of the SMRRs is denoted Smrr.

$$\texttt{Smrr} \triangleq \langle\, \mathit{range} : \wp(\texttt{PhysAddr}), \quad \mathit{strat} : \texttt{CacheStrat} \,\rangle$$

The set of physical addresses *range* tells the core the location of the SMRAM and *strat* indicates which cache strategy has to be used when the core is in SMM.

Cache. Let Index be the set of cache indexes and $\mathit{index} : \texttt{PhysAddr} \to \texttt{Index}$ the function used by the core to determine which index to use for a given physical address. They are both parameters of our model. The cache is divided into several cache lines which contain the cached memory content and several additional information required by the cache strategy algorithm. The set of states of the cache line is denoted by $\texttt{CacheLine}(S)$. In addition to modeling hardware specifications, the definition of $\texttt{CacheLine}(S)$ attaches a software owner to a cache line.

$$\texttt{CacheLine}(S) \triangleq \langle\, \mathit{dirty} : \{\texttt{true}, \texttt{false}\}, \quad \mathit{tag} : \texttt{PhysAddr}, \quad \mathit{content} : \texttt{Val}, \quad \mathit{owner} : S \,\rangle$$

[6] See page xi.

The cache is modeled as a mapping between address indexes and cache lines.

$$\texttt{Cache}(\texttt{S}) \triangleq \texttt{Index} \to \texttt{CacheLine}(\texttt{S})$$

A cache $\texttt{c} \in \texttt{Cache}(\texttt{S})$ is well-formed if every cache line is tagged with a physical address whose index corresponds to the cache line index, that is

$$\forall \texttt{i} \in \texttt{Index}, \mathit{index}(\texttt{c}(\texttt{i}).\mathit{tag}) = \texttt{i}$$

Memory Controller. The set of states of the memory controller is denoted by MC.

$$\texttt{MC} \triangleq \langle\, \mathit{d\_open} : \{\,\texttt{true}, \texttt{false}\,\}, \quad \mathit{d\_lock} : \{\,\texttt{true}, \texttt{false}\,\} \,\rangle$$

The two booleans *d_open* and *d_lock* model two bits of a configuration register named smramc. They are used to determine how the memory controller dispatches the I/O which targets a physical address of the SMRAM. For a memory controller state $\texttt{mc} \in \texttt{MC}$ to be consistent with respect to the hardware specifications, it has to verify that

$$\texttt{mc}.\mathit{d\_lock} = \texttt{true} \Rightarrow \texttt{mc}.\mathit{d\_open} = \texttt{false}$$

We model the SMRAM with two ranges of addresses:

- $\texttt{hSmram} \triangleq \{\,\textsc{Dram}(\texttt{i}) \mid \texttt{smram\_base} \leqslant \texttt{i} \leqslant \texttt{smram\_end}\,\}$ the SMRAM memory range within the DRAM memory
- $\texttt{pSmram} \triangleq \{\,\textsc{Pa}(\texttt{i}) \mid \texttt{smram\_base} \leqslant \texttt{i} \leqslant \texttt{smram\_end}\,\}$ the projection of the SMRAM in the address space manipulated by the core

The values of smram_base and smram_end are specified in the memory controller specifications and are left as a parameter of our model. It is the software responsability to set the SMRR accordingly. We assume $\texttt{smram\_end} - \texttt{smram\_base} > \texttt{0x8000}$, that is the first instruction executed by the core after entering SMM (located at $\texttt{SMBASE} + \texttt{0x8000}$) is inside the SMRAM if the SMBASE register is correctly configured.

The memory controller translates physical addresses into hardware addresses and forwards the I/O accordingly. We model this translation with the function

$$\mathit{dispatch} : \texttt{MC} \times \{\,\texttt{true}, \texttt{false}\,\} \times \texttt{PhysAddr} \to \texttt{HardAddr}$$

Definition 5.2
The function *dispatch* is defined as follows:

$$dispatch(\text{mc}, \text{in\_smm}, \text{pa}) \triangleq \begin{cases} \textsc{Vga}(\text{i}) & \text{if in\_smm} = \texttt{false}, \text{pa} \in \texttt{pSmram}, \\ & \quad \text{and mc.}d\_open = \texttt{false} \\ \textsc{Dram}(\text{i}) & \text{otherwise} \end{cases}$$

$$\text{where } \text{pa} = \textsc{Pa}(\text{i}).$$

We emphasize that the same physical address can be translated into two different hardware addresses for two memory controller states m and m', hence it is possible to have

$$dispatch(\text{m}, \text{b}, \text{pa}) \neq dispatch(\text{m}', \text{b}, \text{pa})$$

Memories. The physical memories (exposed by the DRAM and the VGA controllers) are modeled together with a mapping between the hardware addresses and both their contents and the software components which own them. We write $\texttt{Mem}(\text{S})$ for the set of states of the physical memories.

$$\texttt{Mem}(\text{S}) \triangleq \texttt{HardAddr} \to \langle\ content : \texttt{Val}, \quad owner : \text{S}\ \rangle$$

5.1.3 Transition Labels and Transition Relation

In accordance with Definition 4.1, we consider two sets of labels to distinguish between software transitions (direct, foreseeable consequences of instruction execution, denoted by L_S) and hardware transitions (denoted by L_H). For each transition label we define (1) a precondition to determine whether the transition can occur from a given hardware state, and (2) a postcondition to specify the consequences of the transitions over the hardware architecture state. Let *pre* be a predicate on $\text{L} \times \text{H}(\text{S})$, and *post* be a predicate on $\text{L} \times \text{H}(\text{S}) \times \text{H}(\text{S})$, then the transition relation of Minx86 is defined as follows:

$$\text{h} \xrightarrow[context]{\text{l}} \text{h}' \triangleq pre(\text{l}, \text{h}) \wedge post(\text{l}, \text{h}, \text{h}')$$

Software Transitions. Table 5.1 lists the labels dedicated to software transitions in terms of constructors. We model the core I/Os with *Read*(pa) and *Write*(pa), the configuration of the memory controller with *OpenBitFlip* and *LockSmramc*, the configuration of the cache strategy with *SetCacheStrat*(pa, strat), the configuration of the SMRR with *UpdateSmrr*(smrr), the exit of the SMM with *Rsm*, and the update of the core program counter register with *NextInstruction*(pa).

We now give a description of the pre and postconditions for each label. The interested readers can refer to the Coq development [37] in case they want to review their definitions.

Because the pagination and segmentation mechanisms are outside of the current scope of Minx86, we consider that a core can always read and write at any physical address. As a

Event	Parameters	Description
Write	pa ∈ PhysAddr v ∈ Val	A core I/O to write to physical address pa the value v
Read	pa ∈ PhysAddr	A core I/O to read from physical address pa.
SetCacheStrat	pa ∈ PhysAddr strat ∈ CacheStrat	Change the cache strategy for pa to strat
UpdateSmrr	smrr ∈ Smrr	Update the SMRR content with the new value smrr
Rsm	—	The core leaves SMM
OpenBitFlip	—	Flip the d_open bit
LockSmramc	—	Set the d_lock bit
NextInstruction	pa ∈ PhysAddr	The program counter register of the core is set to *pa*

Table 5.1: List of labels dedicated to Minx86 software transitions (L_S)

consequence, the precondition for *Read*(pa) and *Write*(pa) always holds true. The postconditions for *Read*(pa) and *Write*(pa, v) take into account (1) the cache strategy configured for the address pa, (2) the state of the cache line associated to the index $i = index(pa)$, (3) whether pa belongs to the SMRAM or not, (4) whether the core is in SMM or not, and (5) whether the D_OPEN bit of the SMRAMC register is set or not, in order to determine which memory locations are updated and modify their respective owner accordingly. We discuss two specific cases of the transition labeled *Write*(v, *pa*) from a state h to a state h′ in order to illustrate the definition of the postcondition predicate:

1. If the cache strategy for the address pa is set to WB, the content of this address is not currently cached by the processor, the cache line indexed by $i = index(pa)$ is marked as "dirty", and pa does not fall in the range specified by the core SMRR, that is

$$\begin{aligned} & h.core.strat(pa) = \texttt{WB} \\ \wedge\ & \neg cache\_miss(h.cache, pa) \\ \wedge\ & pa \notin h.smrr.range \\ \wedge\ & h.cache(i).dirty = \texttt{true} \end{aligned}$$

   Then, the content of the cache line is written back to memory and the cache line is updated with v as its new content, $context(h)$ as its new owner, pa as its new tag, and its dirty bit is set. To know which memory location is associated with the address pa, we rely on the *dispatch* function (see Definition 5.2) which models the I/O resolution mechanism of the memory controller. For this specific case, with ha =

Event	Description
ReceiveSMI	A SMI is raised and the core enters SMM
Fetch	A core I/O to fetch the instruction stored at the physical address contained in the program counter register

Table 5.2: List of labels dedicated to Minx86 hardware transitions ($\mathrm{L_H}$)

dispatch(h.*mc*, h.*in_smm*, h.*cache*(i).*tag*), the postcondition is

$$\mathtt{h}' = \mathtt{h}\left\{\begin{array}{c} \mathit{mem}(\mathtt{ha}) \leftarrow \left\langle \begin{array}{c} \mathit{content} = \mathtt{h}.\mathit{cache}(\mathtt{i}).\mathit{content}, \\ \mathit{owner} = \mathtt{h}.\mathit{cache}(\mathtt{i}).\mathit{owner} \end{array} \right\rangle, \\ \mathit{cache}(\mathtt{i}) \leftarrow \left\langle \begin{array}{c} \mathit{content} = \mathtt{v}, \mathit{tag} = \mathtt{pa}, \\ \mathit{owner} = \mathit{context}(\mathtt{h}), \mathit{dirty} = \mathtt{true} \end{array} \right\rangle \end{array}\right\}$$

Note that the same scenario applies if pa belongs to the range specified by the core SMRR, and the cache strategy specified by the SMRR is `WB`.

2. If we consider a similar case, but this time the address pa is already present inside the cache, then only the cache is updated. The new owner of the selected cache line is the software component currently executed by the core, its new content is $\mathtt{v}$ —we remind the considered transition is *Write*($\mathtt{v}$, pa)— and its dirty bit is set, while the tag of the cache line remains untouched. In this second scenario, the postcondition is

$$\mathtt{h}' = \mathtt{h}\left\{ \mathit{cache}(\mathtt{i}) \leftarrow \mathtt{h}.\mathit{cache}(\mathtt{i})\left\{\begin{array}{c} \mathit{content} = \mathtt{v}, \\ \mathit{owner} = \mathit{context}(\mathtt{h}), \\ \mathit{dirty} = \mathtt{true} \end{array}\right\}\right\}$$

A software component can always update the cache strategy used for an I/O. The postcondition for *SetCacheStrat*(pa, *strat*) requires only the cache strategy setting for this physical address pa to change. The precondition for *UpdateSmrr* requires the core to be in SMM. The postcondition requires the SMRR of the core to be updated with the correct value, the rest of the hardware architecture state being left unchanged.

A software component can jump to any physical address, that is the preconditions for the label *NextInstruction*(pa) always hold true. The postcondition for *NextInstruction*(pa) requires the program counter register to be updated with pa. The *OpenBitFlip* precondition requires the `SMRAMC` register to be unlocked. The postcondition requires the `d_open` bit to be updated. The *LockSmramc* precondition requires the SMRAM to be unlocked, *i.e.* the `d_lock` bit to be unset. The postcondition requires the `d_open` bit to be unset and the `d_lock` bit to be unset.

Hardware Transitions. Table 5.2 lists the labels dedicated to hardware transitions in terms of constructors.

ReceiveSMI models a SMI being raised and handled by the core. Its precondition requires the core not to be in SMM because SMM is non-reentrant. The postcondition of *ReceiveSMI* requires

$$\frac{\mathtt{h}.\mathit{core.in\_smm} = \mathtt{false} \qquad \mathtt{h}' = \mathtt{h}\left\{\begin{array}{c}\mathit{core.in\_smm} \leftarrow \mathtt{true},\\ \mathit{core.pc} \leftarrow \mathtt{h}.\mathit{core.smbase} + \mathtt{0x8000}\end{array}\right\}}{\mathtt{h} \xrightarrow[\mathit{context}]{\mathit{ReceiveSMI}} \mathtt{h}'}$$

Figure 5.1: Pre and postconditions for MINX86 *ReceiveSMI* transitions

the program counter to be set to $\mathtt{SMBASE} + \mathtt{0x8000}$ and the core enters in SMM. Both the pre and postcondition of *ReceiveSMI* are formally defined in Figure 5.1. *Fetch* models the I/O to fetch the instruction pointed by the program counter register. As a consequence, the definition of its precondition and postcondition are the same as $\mathit{Read}(\mathtt{h}.\mathit{core.pc})$. *Fetch* is a hardware transition because the address used to fetch the next instruction is determined by the program counter register of the core. A software component can modify the value of this register —this is modeled by *NextInstruction*— but it is not alone in this case. The hardware architecture will update the register independently to the software component it currently executes, *e.g.* when it receives an interrupt.

5.1.4 Transition-Software Mapping

We define MINX86_*fetched* a transition-software mapping for MINX86 (Definition 4.10), to map an initial state of a transition and the label of this transition to the set of software components which own the instruction fetched during this transition. In the case of MINX86, there is only one event which implies fetching instructions: *Fetch*. To define the transition-software mapping in this specific case, we rely on a helper function $\mathit{read\_from\_cache} : \mathcal{H}(S) \times \mathtt{PhysAddr} \to \{\mathtt{true}, \mathtt{false}\}$ which returns `true` if a core will read the content of a given physical address from the cache, and `false` otherwise. This function is defined as follows:

$$\mathit{read\_from\_cache}(\mathtt{h}, \mathtt{pa}) \triangleq \left\{\begin{array}{ll}\mathtt{true} & \text{if } \mathtt{pa} \notin \mathtt{h}.\mathit{core.smrr.range},\ \mathtt{h}.\mathit{core.strat} = \mathtt{WB}\\ & \quad \text{and } \mathit{cache\_hit}(\mathtt{h}.\mathtt{cache}, \mathtt{pa})\\ \mathtt{true} & \text{if } \mathtt{pa} \in \mathtt{h}.\mathit{core.smrr.range},\ \mathtt{h}.\mathit{core.smrr.strat} = \mathtt{WB},\\ & \quad \mathit{cache\_hit}(\mathtt{h}.\mathtt{cache}, \mathtt{pa}) \text{ and } \mathtt{h}.\mathit{core.in\_smm} = \mathtt{true}\\ \mathtt{false} & \text{otherwise}\end{array}\right.$$

Using the function *read_from_cache*, we can define the transition-software mapping of MINX86 as follows:

$$\text{MINX86\_}\mathit{fetched}(\mathtt{h}, \mathtt{l}) = \left\{\begin{array}{ll}\emptyset & \text{if } \mathtt{l} \neq \mathit{Fetch}\\ \emptyset & \text{if } \mathtt{h}.\mathit{core.pc} \in \mathtt{h}.\mathit{core.smrr.range}\\ & \quad \text{and } \mathtt{h}.\mathit{core.in\_smm} = \mathtt{false}\\ \{\mathtt{h}.\mathit{cache}(\mathit{index}(\mathtt{pc})).\mathit{owner}\} & \text{where } \mathtt{pc} = \mathtt{h}.\mathit{core.pc} \text{ and}\\ & \quad \text{if } \mathit{read\_from\_cache}(\mathtt{h}, \mathtt{pc}) = \mathtt{true}\\ \{\mathtt{h}.\mathit{mem}(\mathtt{ha}).\mathit{owner}\} & \text{where}\\ & \quad \mathtt{ha} = \mathit{dispatch}(\mathtt{h}.\mathit{mc}, \mathtt{h}.\mathit{in\_smm}, \mathtt{h}.\mathit{core.pa})\end{array}\right.$$

Firstly, an instruction is fetched during a *Fetch* transition only, so for other transitions, the function returns the empty set. Secondly, if a core not in SMM tries to read the content of a physical addresses which belongs to the range specified by its SMRR, the specification states that the result of this I/O is a constant value, hence the function also returns the empty set in this case. For the last two cases, we based our decision on the *read_from_cache* function to determine the content of the singleton returned by Minx86_*fetched*.

In this Section, we have defined a minimal x86 hardware model we called Minx86. In its current state, the scope of Minx86 is limited to hardware features involved in the HSE mechanism implemented by the BIOS to remain isolated from the rest of the software stack at runtime. However, we have defined Minx86 with extensibility in mind. Notably, the hardware-software mapping used by the model in order to track memory location ownership is left as a parameter, and Minx86 could be extended to reason about other HSE mechanisms whose purpose is to enforce a code injection policy.

5.2 Specifying and Verifying a BIOS HSE Mechanism

We consider the execution of the software stack made of a BIOS, an operating system and n applications, after the end of the boot sequence. In this Section, our objective is to implement the three-step methodology that we have introduced in Chapter 4, that is (1) specifying the software requirements that must be satisfied by the trusted software components which implement the HSE mechanism, (2) specifying the targeted security policy the HSE mechanism supposedly enforces, and (3) verifying that the HSE mechanism is correct with respect to the targeted security policy.

Since the targeted security policy has already been defined previously (Definition 4.14), it leaves steps (1) and (3) to complete. We first give a formal definition of the HSE mechanism implemented by the BIOS (denoted by $\Delta_{\texttt{bios}}$) to remain isolated from the rest of the software stack (5.2.1). Then, we verify its correctness with respect to the BIOS code injection sub-policy $\mathrm{I}_{\texttt{bios}}$ (Definition 4.14) (5.2.2). We conclude this Section with an overview of the approach we used to organize our machine-checked proof, written in Coq (5.2.3)

5.2.1 BIOS HSE Definition

Minx86 is parameterized with a hardware-software mapping *context* (Definition 4.3). This mapping is necessary for the model to track memory ownership during *Write*, *Read* and *Fetch* transitions. Because Minx86 is not precise enough to distinguish between the execution of the operating system and the applications, we axiomatize the definition of *context* with two rules:

context-**bios:** If the core is in SMM, it executes the BIOS

$$\mathrm{h}.\mathit{in\_smm} = \texttt{true} \Rightarrow \mathit{context}(\mathrm{h}) = \texttt{bios}$$

context-**untrusted:** If the core is not in SMM, it executes another component of the software stack

$$\mathtt{h}.\mathit{in\_smm} = \mathtt{false} \Rightarrow \mathit{context}(\mathtt{h}) \in \{\, \mathtt{os}, \mathtt{app}_1, \ldots, \mathtt{app}_n \,\}$$

We define $\Delta_{\mathtt{bios}}$ to model the HSE mechanism applied by the BIOS, such that

$$\Delta_{\mathtt{bios}} = \langle S, \{\, \mathtt{bios} \,\}, \mathit{context}, \mathit{hardware\_req}, \mathit{bios\_req} \rangle$$

As *context* has already been specified, only *hardware_req* and *bios_req* remain. Once we have properly introduced both, we verify our definition of $\Delta_{\mathtt{bios}}$ satisfies the HSE laws (Definition 4.5).

Requirements over States. Given $\mathtt{h} \in H(S)$, we have identified six requirements over states for $\mathtt{h}$ to satisfy *hardware_req*.

1. When the core executes the SMM code, the program counter register value needs to be an address in SMRAM.
$$\mathit{context}(\mathtt{h}) = \mathtt{bios} \Rightarrow \mathtt{h}.\mathit{core.pc} \in \mathtt{pSmram}$$
2. The SMBASE register was correctly set during the boot sequence to point to the base of the SMRAM.
$$\mathtt{h}.\mathit{core.smbase} = \textsc{Pa}(\mathtt{smram\_base})$$
3. All the memory locations within the SMRAM are owned by the BIOS.
$$\forall \mathtt{ha} \in \mathtt{hSmram}, \mathtt{h}.\mathit{mem}(\mathtt{ha}).\mathit{owner} = \mathtt{bios}$$
4. For a physical address in SMRAM, in case of cache hit, the related cache line content must be owned by the SMM code.
$$\forall \mathtt{pa} \in \mathtt{pSmram}, \mathit{cache\_hit}(\mathtt{h}.\mathit{cache}, \mathtt{pa}) \Rightarrow \mathtt{h}.\mathit{cache}(\mathit{index}(\mathtt{pa})).\mathit{owner} = \mathtt{bios}$$
5. In order to protect the content of the SMRAM inside the DRAM memory, the boot sequence code has locked the SMRAMC controller. This ensures that an OS cannot set the d_open bit any longer and only a core in SMM can modify the content of the SMRAM.
$$\mathtt{h}.\mathit{mc.d\_lock} = \mathtt{true}$$
6. The range of memory declared with the SMRR needs to overlap with the SMRAM.
$$\mathtt{pSmram} \subseteq \mathtt{h}.\mathit{core.smrr.range}$$

During the boot sequence, the BIOS is the only software component executed, so there is no untrusted software component whose execution needs to be constrained. It is expected that the BIOS configures the hardware architecture in a way which satisfies the requirements over states prior to start the execution of the rest of the software stack. Otherwise, the runtime will not start from a "safe" state and the related traces will not comply to $\Delta_{\mathtt{bios}}$ (Definition 4.6). In such a case, there is no guarantee that the security policy will be enforced, independently of the correctness of $\Delta_{\mathtt{bios}}$.

Requirements over Transitions. We now define *bios_req*. We only define two restrictions. First, we force the BIOS execution to remain confined within the SMRAM. The reason is simple: the OS can tamper with the memory outside the SMRAM. Secondly, we prevent the BIOS to update the SMRR registers. These registers should have been properly configured during the boot sequence, and there is no need to further update them.

$$
\begin{aligned}
\mathit{bios\_req}(\mathtt{h},\mathtt{l}) \triangleq\ & \mathit{context}(\mathtt{h}) = \mathtt{bios} \\
& \Rightarrow ((\forall \mathrm{pa} \in \mathtt{PhysAddr}, \\
& \qquad \mathtt{l} = \mathit{NextInstruction}(\mathrm{pa}) \Rightarrow \mathrm{pa} \in \mathtt{pSmram}) \\
& \quad \wedge (\forall \mathrm{smrr} \in \mathtt{Smrr}, \mathtt{l} \neq \mathit{UpdateSmrr}(\mathrm{smrr})))
\end{aligned}
$$

Verifying the HSE Laws. For $\Delta_{\mathtt{bios}}$ to be a HSE mechanism, we need to prove it satisfies the two HSE laws.

The first law states that *bios_req* is always satisfied when a non-trusted software component, i.e. different from the BIOS, is executed by the hardware architecture. By definition of *bios_req*, $\mathit{context}(\mathtt{h}) = \mathtt{bios}$ is an antecedent of the requirements over *NextInstruction* and *UpdateSmrr*. Therefore, the first HSE law is satisfied for $\Delta_{\mathtt{bios}}$.

The second law requires the state requirements to be invariant with respect to the software requirements. We prove this by case enumeration of $\mathtt{l} \in \mathrm{L_S} \uplus \mathrm{L_H}$ and $\mathtt{h} \in \mathrm{H}(S)$. We check that each requirement described previously is preserved by *bios_req*. In practice, these proofs turned out to be the more demanding, especially for requirement 3. —the content of the SMRAM is owned by the BIOS— and 4. —the cache line tagged with physical addresses related to the SMRAM are owned by the BIOS— because this requires to take the SMRR and the write-back strategy into account for the *Write* and *Read*. We take the example of *Read* transitions; a similar approach applies to *Write* transitions.

Given a transition labeled $\mathit{Read}(\mathrm{pa})$ from $\mathtt{h}$ to $\mathtt{h}'$, there are three cases to consider:

1. The core discards the read, as part of the SMRAM cache poisoning countermeasure. In this context, the hardware architecture is not updated, therefore $\mathtt{h} = \mathtt{h}'$ and we can conclude that $\mathit{hardware\_req}(\mathtt{h}) \Rightarrow \mathit{hardware\_req}(\mathtt{h}')$.

2. The core is configured to read the content of $\mathtt{pa}$ from the underlying memories (uncacheable strategy). Thanks to requirements 5., we know that the SMRAMC register has been correctly configured, so by definition of *dispatch* (Definition 5.2), we know that only a core in SMM can update the content of the SMRAM. We conclude that the content of the SMRAM remains owned by the BIOS in $\mathtt{h}'$.

3. The core is configured to read the content of $\mathtt{pa}$ from the cache, and may have to perform a cache eviction if necessary (write-back strategy). This last case is the more complex, because we have to consider: whether the core is in SMM or not; whether the targeted physical address falls into the SMRR range or not; whether the I/O results in a cache hit or not; in case of cache miss, whether the occupied cache line is tagged "dirty" or not; in case of a dirty bit, whether its tag (a physical address) belongs to the SMRAM or not.

Taken separately, each case is relatively straightforward to prove. For instance, in case of cache miss, but the occupied cache line is not tagged as "dirty", then the underlying memories are not updated, so we can conclude that the SMRAM remains owned by the BIOS. To manage the number of cases to consider, we carefully organize our proofs in general-purpose lemmas which applies to both *Read* and *Write* transitions[7].

Once each requirement has been shown to be preserved during a transition, we can conclude for *hardware_req* as a whole.

5.2.2 BIOS HSE Mechanism Correctness

Our objective is to prove that $\Delta_{\texttt{bios}}$ is correct with respect to $\mathrm{I}_{\texttt{bios}}$. Because $\mathrm{I}_{\texttt{bios}}$ is a security policy modeled with a predicate on transitions, we know thanks to Theorem 4.1 that

$$\begin{aligned}&\forall(\mathrm{h},\mathrm{l},\mathrm{h}') \in \mathcal{T}(\textsc{Minx86}(\mathit{context})),\\ &\quad(\mathit{hardware\_req}(\mathrm{h}) \wedge (\mathrm{l} \in \mathrm{L_S} \Rightarrow \mathit{bios\_req}(\mathrm{h},\mathrm{l}))) \Rightarrow \mathrm{I}_{\texttt{bios}}(\mathrm{h},\mathrm{l},\mathrm{h}')\end{aligned}$$

is a sufficient condition for $\Delta_{\texttt{bios}} \models \mathrm{I}_{\texttt{bios}}$.

From the security policy perspective, only the *Fetch* label is relevant. Indeed, a code injection can only occur when an instruction owned by a software components is fetched in order to be executed by a core (Definition 4.2.1). For Minx86, this only happens during transitions labeled by *Fetch* by definition of Minx86_*fetched* (see Subsection 5.1.4). *Fetch* is a hardware transition, and therefore is not concerned by *bios_req*. As a consequence, we can simplify our proof goal as follows:

$$\begin{aligned}&\forall(\mathrm{h},\mathrm{h}') \in \mathrm{H(S)} \times \mathrm{H(S)} \text{ such that } \mathrm{h} \xrightarrow[\mathit{context}]{\mathit{Fetch}} \mathrm{h}',\\ &\quad\mathit{hardware\_req}(\mathrm{h}) \Rightarrow \mathrm{I}_{\texttt{bios}}(\mathrm{h},\mathit{Fetch},\mathrm{h}')\end{aligned}$$

We now unfold the definition of $\mathrm{I}_{\texttt{bios}}$ (Definition 4.14) and of the code injection (Definition 4.11), which allows us to simplify further our proof goal as follows:

$$\begin{aligned}&\forall(\mathrm{h},\mathrm{h}') \in \mathrm{H(S)} \times \mathrm{H(S)} \text{ such that } \mathrm{h} \xrightarrow[\mathit{context}]{\mathit{Fetch}} \mathrm{h}',\\ &\quad(\mathit{hardware\_req}(\mathrm{h}) \wedge \mathit{context}(\mathrm{h}) = \texttt{bios}) \Rightarrow \textsc{Minx86}\_\mathit{fetched}(\mathrm{h},\mathit{Fetch}) = \{\,\texttt{bios}\,\}\end{aligned}$$

The rest of the proof leverages the definition of *hardware_req*. Because of requirements 1., we know that $\mathrm{h}.\mathit{core}.\mathit{pc}$ belongs to the `pSmram`. By definition of Minx86_*fetched*, we know that we have two cases to consider:

– The instruction is read from the cache, and therefore

$$\textsc{Minx86}\_\mathit{fetched}(\mathrm{h},\mathit{Fetch}) = \{\,\mathrm{h}.\mathit{cache}(\mathit{index}(\mathrm{h}.\mathit{core}.\mathit{pc})).\mathit{owner}\,\}$$

In this context, we can conclude using the requirements 4.

[7]The interesting reader can have a look at the file `src/Smm/Delta/Preserve/Architecture.v` in [37], which gather these proofs, and `src/Smm/Delta/Preserve/{Read,Write}.v` to see them in action.

– Otherwise, the instruction is read from the underlying memories. More precisely, by the definition of *dispatch* (Definition 5.2), we know the instruction is read from the DRAM, and as a consequence the requirements 3. allows us to conclude.

As a side note, we emphasize that these proofs validate the SMRAM cache poisoning countermeasure. Without the SMRRs, it is not possible to conclude about the correctness of $\Delta_{\texttt{bios}}$ if the hardware model takes the cache into account. We consider two scenarios. On the one hand, the HSE mechanism definition takes the cache into account too. In these conditions, it is not possible to prove it satisfies the second law. Indeed, the cache poisoning attack violates the requirement 4. On the other hand, the HSE mechanism definition does not take the cache into account. As a consequence, it is possible to prove it satisfies the second law. However, it is not possible to prove it is correct with respect to $\mathrm{I}_{\texttt{bios}}$. Indeed, we cannot discard the case where a cache line tagged with a physical address which belongs to the SMRAM is owned by an untrusted software component. In this scenario, we could even exhibit a trace which complies to the HSE mechanism, but does not satisfy the security policy.

5.2.3 On SpecCert Machine-Checked Proofs

As we already explained in the introduction, we have implemented the model and proofs of this Chapter in Coq, to increase our confidence in our results. The resulting project [37] counts around 4 500 lines of code divided into 2 000 lines of code of definition and specification and 2 500 lines of code of proofs. We have taken great care to organize our development in order to manage the many sub-cases notably induced by the write-back strategy.

Each hardware component has been defined in isolation from the rest of the hardware architecture, with a set of functions, a set of properties and a set of general-purpose lemmas. For instance, the directory `src/Cache/` contains four files: `Cache_def.v`, `Cache_func.v`, `Cache_proofs.v`, and `Cache_prop.v`. We then have defined $\mathrm{H}(\mathrm{S})$, that is the Cartesian product of individual components, in a similar manner. Inside the `src/x86/Architecture/` directory, we defined general-purpose functions (`Architecture_func.v`) and lemmas (`Architecture_proofs.v`) to reason about the hardware architecture as a whole as often as possible.

This approach has proven to be valuable to deal with the many cases to cover when used in conjunction with a recurring pattern we call "`remember`; `destruct`; `assert`; `apply`", after the Coq tactics. We illustrate the use of this pattern with the proof goal pictured in Figure 5.2.

1. Using `remember`, we can make the sequence of intermediary state updates more visible (see Figure 5.3, compared to Figure 5.2). Each `a_i` is an intermediary state resulting in a call to a general-purpose function from the `Architecture_func.v` file.

2. Using `destruct`, we can explore the alternative paths resulting from the use of Gallina constructions such as pattern matching, or `if-then-else`. For instance, regarding the initial state of the system, the content of the address pa may or may not be present in the cache (Figure 5.3, line 14). We explore both alternatives using the `destruct` tactic (see the two simplified goals in Figure 5.4, more particularly on the value of `a_3`).

```
1  Hreq : hardware_req a
2  ============================================================================
3  hardware_req
4      (update_cache_content
5         (if cache_hit_dec (cache a) pa
6          then a
7          else
8            update_cache_content
9              (if cache_location_is_dirty_dec (cache a) pa
10              then
11                update_memory_content
12                  a
13                  (phys_to_hard a (cache_location_address (cache a) pa))
14                  (find_in_cache_location (cache a) pa)
15             else a)
16            pa
17            (find_memory_content a (phys_to_hard a pa))) pa
18       (context (proc a)))
```

Figure 5.2: Raw postcondition of a `Write` transition with a writeback strategy

3. Using `assert`, we can introduce new goals to prove intermediary results (like the one in Figure 5.5, for instance). In our case, we will prove that the predicate `hardware_req` remains satisfied after each state update.

4. Using `apply`, we can leverage lemmas about `hardware_req` preservation for a given state update, *e.g.* exported by the `Architecture_proofs.v` or `Cache_proofs.v` files. For instance, the function `update_cache_content` (Figure 5.3, line 11) has a companion lemma called `update_cache_content_with_context_preserves_inv` that we can use to conclude about the goal introduced in Figure 5.5.

Figure 5.6 depicts the typical shape of resulting proof trees. The proofs are structured around smaller, more atomic proof goals of the form $\iota(h_i) \Rightarrow \iota(h_{i+1})$, where ι is supposedly an invariant, h_i is an intermediary state, and h_{i+1} its successor.

The resulting "proof strategy" has proven to be effective to manage the relative complexity of our proof goals. However, it remains cumbersome to use manually, because it requires to generate *manually* an important number of hypotheses (*via* one per `remember` and `assert` use), which sometimes increases significantly the size of the proof obligation generated by Coq.

```
Hreq : hardware_req a
Heqa_2 : a_2 =
         update_memory_content
           a
           (phys_to_hard a (cache_location_address (cache a) pa))
           (find_in_cache_location (cache a) pa)
Heqa_3 : a_3 = (if cache_location_is_dirty_dec (cache a) pa
                then a_2
                else a)
Heqa_4 : a_4 =
         update_cache_content a_3
                              pa
                              (find_memory_content a (phys_to_hard a pa))
Heqa_5 : a_5 = (if cache_hit_dec (cache a) pa
                then a
                else a_4)
Heqa_6 : a_6 = update_cache_content a_5 pa (context (proc a))
============================================================================
hardware_req a_6
```

Figure 5.3: Postcondition of a `Write` transition with a writeback strategy, after the use of the `remember` tactic

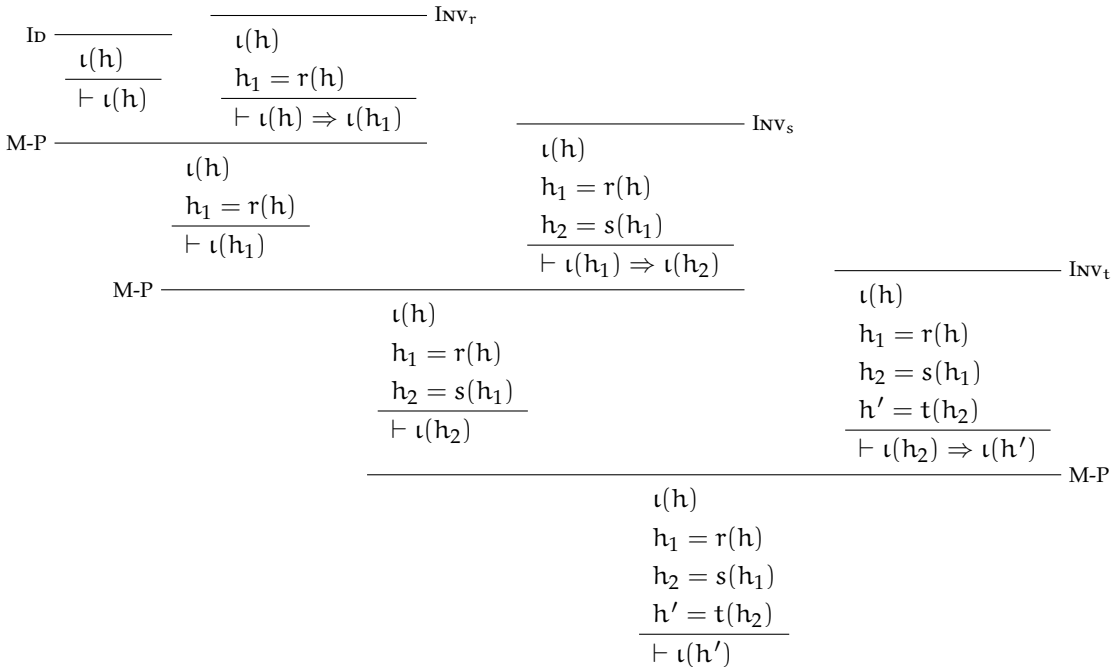
```
(* -- >8 -- *)
Hc : cache_location_is_dirty (cache a) pa
Heqa_3 : a_3 = a_2
(* -- >8 -- *)
============================================================================
hardware_req a_6
```

```
(* -- >8 -- *)
Hc : ~ cache_location_is_dirty (cache a) pa
Heqa_3 : a_3 = a
(* -- >8 -- *)
============================================================================
hardware_req a_6
```

Figure 5.4: Exploring alternative paths using the `destruct` tactics

```
1  Hreq : hardware_req a
2  (* -- >8 -- *)
3  Heqa_4 : a_4 =
4           update_cache_content a_3
5                                pa
6                                (find_memory_content a (phys_to_hard a pa))
7  (* -- >8 -- *)
8  ============================================================================
9  hardware_req a_4
```

Figure 5.5: Intermediary statements, generated using `assert` tactics



where M-P is the *Modus ponens*, and $\textsc{Inv}_r$, $\textsc{Inv}_s$, and $\textsc{Inv}_t$ are intermediary lemmas previously proved true.

Figure 5.6: Dividing a transition into sequences of intermediary states updates

5.3 Conclusion

In this Chapter, we have proceeded with the case study introduced in Section 4.2. We introduced MINX86, a minimal x86 hardware model conceived with extensibility in mind. We specified the HSE mechanism implemented by the BIOS in order to remain isolated from the rest of the software stack, then verified that this mechanism is correct with respect to the so-called BIOS code injection sub-policy. In summary, we have applied our three-step methodology. The resulting model assumes as little as possible about the actual implementation of the BIOS, and constitutes, to the extent of our knowledge, the first formalization of the BIOS security model at runtime. As we have already mentioned, our proofs have been implemented within the Coq theorem prover to increase our confidence in our results, and have been released as free software [37]. The resulting project counts around 4 500 lines of code, and we believe this represents a manageable size regarding the complexity of the model, but whether our approach remains applicable for a large-scale models remains to be proven when we first released this work in 2016. Subsequently, we have experimented increasing the scope of MINX86 with pagination, and have been convinced that considering the whole architecture at once —as it is done in MINX86— would not scale properly.

The next Part of this thesis focuses on this challenge. We propose another approach to enable modular verification of complex systems composed of interconnected components, such as hardware architectures.

Part III

Towards Comprehensive Hardware Models

6

Modular Verification of Component-based Systems

"Give someone a program, you frustrate them for a day; teach them how to program, you frustrate them for a lifetime."

— David Leinweber

In the previous Part of this manuscript, we have introduced a theory of HSE mechanisms and have leveraged it to specify and verify the HSE mechanism implemented by the BIOS at runtime in order to remain isolated from the rest of the software stack. In this context, we have defined a dedicated x86 hardware model called Minx86 (Definition 5.1). This experiment has been very insightful. In particular, it has highlighted the key importance of a modular approach to tackle the challenge posed by the scale of the x86 hardware architecture.

This Chapter focuses on our second contribution: an approach to model and verify each component of a component-based system in isolation, while providing the necessary abstractions to compose these components [16], along with a companion framework for Coq we called FreeSpec to support this approach [21].

The rest of the Chapter proceeds as follows. We motivate our contribution in accordance with our experience implementing the proofs of Chapter 5 (Section 6.1). Then, we describe how we model components in terms of programs with effects and effect handlers (Section 6.2). Finally, we introduce so-called abstract specifications to verify their respective properties, and discuss two verification strategies which leverage them (Section 6.3).

6.1 Lessons Learned from Minx86

In this Section, we emphasize three limitations of Minx86 which makes its use at a larger scale unlikely (6.1.1). Then, we give an overview of the approach we proposed to address them (6.1.2).

6.1.1 Minx86 Limitations

Having reasoned about the correctness of $\Delta_{\texttt{bios}}$, we have identified three limitations which reduce the applicability of Minx86 as we would increase its scope:

1. its monolithic design;
2. its weakness against the temporary violation problem [129];
3. the presence, within our supposedly general-purpose model, of abstract values that are specific to a particular case study.

Monolithic Model. The set of states $\mathrm{H}(\mathrm{S})$ of Minx86 is the Cartesian product of the sets of states of its hardware components, namely a core, a cache, a memory controller and two arrays of memory cells. That is, Minx86 is a monolithic model, and this is not without consequences.

During the development of our proof of concept, changes to the definition of $\mathrm{H}(\mathrm{S})$ had an important impact on proofs already written at the moment. We have no doubt that such situations will arise again if we try to increase the scope of Minx86.

Besides, some transitions of Minx86 imply several intermediary state updates. This is particularly true for *Read* and *Write* transitions, as both the cache and memories are updated in case of cache misses or cache evictions. The complexity of a transition postcondition has a direct impact on the proofs we have to construct in order to verify the correctness of a given system, as the number of case scenarios to consider increases quickly. We have already given an indication of this in Subsection 5.2.1 when we explained the structure of the proof that $\Delta_{\texttt{bios}}$ complies with the second HSE law. The proof goal depicted in Figure 5.2 —as presented by Coq when we are constructing our proof— is a good illustration of that, considering it is already a subcase of the complete proof. In this small snippet, we can identify four intermediary states (as shown in Figure 5.3).

Temporary Violation Problem. The complexity of Minx86 transitions emphasizes another important limitation: it is potentially subject to the so-called *temporary violation problem* [129], where a predicate is satisfied *before* and *after* a given transition, yet is temporary violated *during* the transition. A well-known illustration of this threat is —once again— the airlock system.

Example 6.1 (Temporary Violation Problem in an Airlock System)
In Example 3.1, we have carefully defined the labeled transition system of the airlock system so that it is unambiguously not subject to the temporary violation problem. As a reminder, the targeted security policy for an airlock system is that at least one door shall be closed at all time. We did that by avoiding transitions between states $(\texttt{open}, \texttt{closed})$ and $(\texttt{closed}, \texttt{open})$. We can define a transition system which is correct with respect to the targeted safety property *and* includes these transitions.

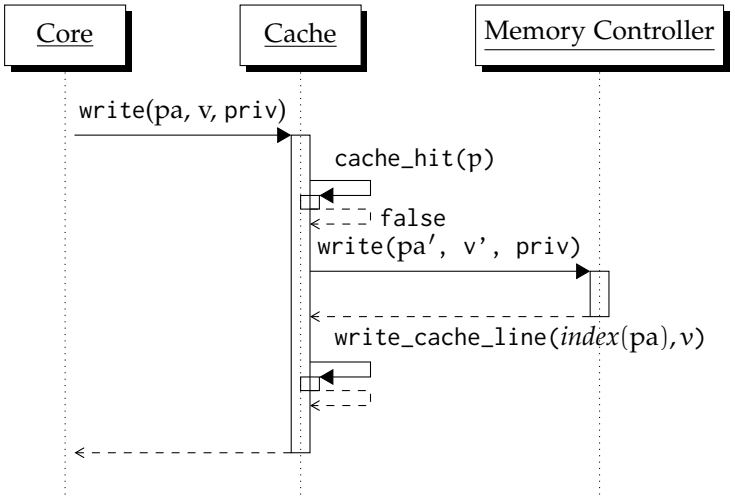
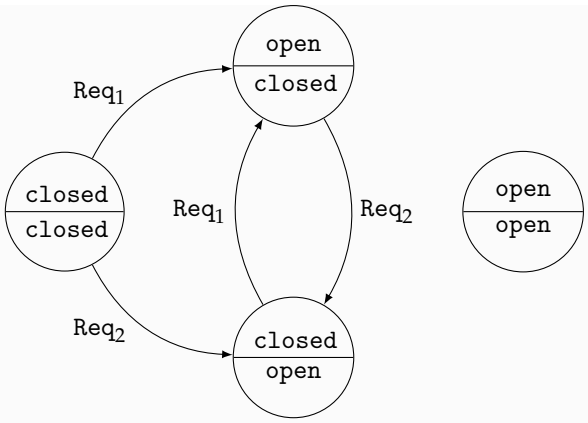


Figure 6.1: Sequence diagram of the execution of `movq $0, (%rax)`



We can imagine several scenarios: (1) Doors states updates are simultaneous and instantaneous; in other words, the transitions *are* atomic. (2) The airlock system first closes one door before opening the other. (3) The airlock system first opens one door before closing the other. The systems modeled in Examples 3.1, 3.4, and 3.5 correspond to the scenario (2). On the contrary, the transition system pictured in this example is not precise enough to discard the eventuality that both doors are, at a given point in time, both open.

The temporary violation problem undermines the second HSE law, which stipulates that the requirement over states of a HSE mechanism is an invariant of the model with respect to the requirement over software transitions (see Definition 4.5). In practice, transitions in MINX86 are not atomic (they are made of several intermediary state updates, as depicted in Figure 6.1). Attackers may be able to leverage a temporary violation of invariant, similarly to the Speed Racer attack [7] detailed in Subsection 2.3.2. Modeling that threat without changing

the modeling structure of Minx86 requires to increase the granularity of the transitions, with the direct consequence to complicate the model.

Abstract Model. Our decision to embed the memory location ownership directly inside Minx86, alongside with the concrete content of each memory location, contradicts our objective to rely on general-purpose models to specify and verify HSE mechanisms. Continuing with a similar logic would mean modifying the model every time we want to reason about a security policy which requires to maintain additional information. For instance, if we want to determine how many times memory locations have been updated since the beginning of the trace, we would have to modify the definitions of $\mathsf{H}(\mathsf{S})$ and of the transition relation to add a counter along with the owner of a memory location. Modifying the transition relation is likely to break all proofs previously written, and therefore this greatly reduces the opportunity to reuse Minx86 for other security policies.

6.1.2 FreeSpec Overview

In the verification work presented in Chapter 5, we have successfully managed the limitations of Minx86 partly because we have taken great care in organizing our development as modurarly as we could. We detailed in Subsection 5.2.3 the strategy we have leveraged to organize our proofs. We have decoupled the hardware components as much as possible, and we leveraged the "`remember`; `destruct`; `assert`; `apply`" pattern to reason about their successive state updates. As a result, the proofs are structured around smaller, more atomic proof goals. That is, our proof strategy has proven to be an effective way to mitigate the monolithic design of Minx86, and has the potential to address the temporary violation problem (thanks to the intermediary sub-case we introduce with the `assert` tactic). However, it remains cumbersome to use manually.

Objectives. With FreeSpec, our objective is twofold.

Firstly, we want to propose a general-purpose formalism that allows for never considering the hardware architecture as a whole, but rather focus on interactions between neighboring components. To that end, we take the principle of our theory of HSE mechanisms a step further. Our approach is based onto a clear separation between hardware components from the one side, and software components from the other side. Software components can interact with hardware components through the execution of instructions, modeled by the software transitions of the hardware model, and we specify requirements over how trusted software requirements interact with the hardware architecture. With FreeSpec, we apply the same reasoning for hardware components alike: we specify how they interact together, and specify requirements over these interactions.

Secondly, we want to automate as much as possible the `remember`; `destruct`; `assert`; `apply` pattern, in a way that enables formally reasoning about the temporary violation problem. This objective is reminiscent of the programming language problematic to model and verify large programs with side effects. Reasoning about side effects in purely functional languages such as

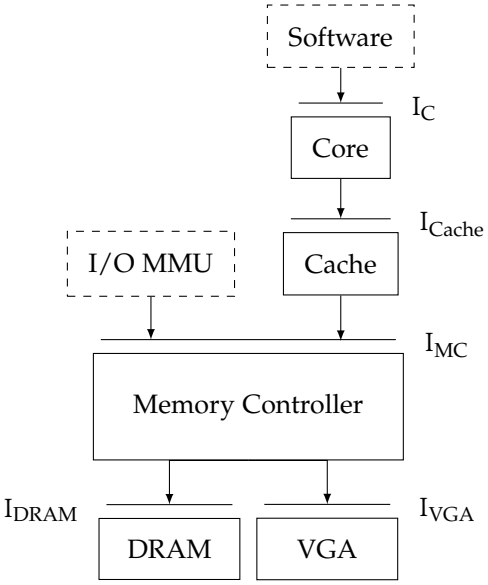


Figure 6.2: Interface-driven modeling of the x86 architecture

Gallina, the Coq specification language, is difficult, firstly because they imply to somehow take into account an outer *stateful* environment and secondly, because the composition of stateful computations is not well handled by traditional (monadic) approaches. Algebraic effects and handlers [24] overcome this double challenge. They allow to model large classes of effects (e.g., exception, state, non-determinism) and to compose effects within purely functional programs, while deferring the realizations of these effects to dedicated handlers. In this Chapter, we aim to show how a variant of algebraic effects based on free monads can be used to support reasoning about component-based systems, including with respect to the temporary violation problem.

We now give an overview of the formalism that we propose to overcome MINX86 limitations, both to model and verify a hardware architecture. Its key concept is to focus on components *interfaces*, where an interface is a set of interdependent operations which are expected to produce a value.

Interface and Component. A component is characterized primarily by the interface it exposes to the rest of the system, and secondarily by its current state and the interfaces it uses in order to operate. A component receives requests to compute results of operations through its interface, and sends computational requests to other components it is connected to *via* their interfaces and waits for their results.

Example 6.2 (Minx86 as a component-based system)
Figure 6.2 pictures an alternative model to MINX86. The software components interact with the core *via* its instruction set. When the core needs to read from or write to memory, it leverages

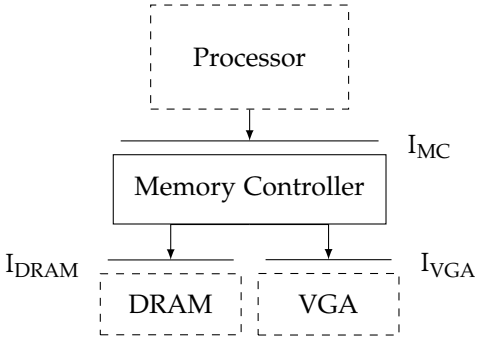


Figure 6.3: The memory controller in isolation

its cache. In case of cache miss or cache eviction, the cache interacts with the system memory through the memory controller, whose goal is to dispatch memory accesses between *e.g.* DRAM and the VGA controller. In addition, the memory controller dispatches memory requests coming from other sources, for instance an I/O MMU —something that we have not taken into account in the proofs in Chapter 6.

Abstract Specifications. To enable compositional reasoning about component-based systems, we introduce so-called abstract specifications which are characterized by requirements over how the interface should be used and requirements over the interface operations results.

Example 6.3 (BIOS Isolation)
We consider one more time the isolation of the BIOS at runtime, thanks to the SMM and the SMRAM.

The `SMRAMC` register, exposed by the memory controller, allows the BIOS to configure the main access control mechanism which protects the content of the SMRAM from the rest of the software stack. Using FreeSpec, we can model and verify this access control mechanism in isolation, by focusing on the memory controller alone. This is achieved by abstracting away the rest of the hardware model, thanks to the interfaces exposed and used by the hardware component of interest, as pictured in Figure 6.3. We model the security policy enforced by the mechanism behind the `SMRAMC` register thanks to an abstract specification. An abstract specification is a couple of requirements —over the interface users, and over the interface results— which translates as follows here:

– We forbid unprivileged I/O targeting the SMRAM when the `SMRAMC` register is not correctly configured.

– We guarantee that a processor which read the content of the SMRAM while in SMM will get what it has previously written to SMRAM while in SMM.

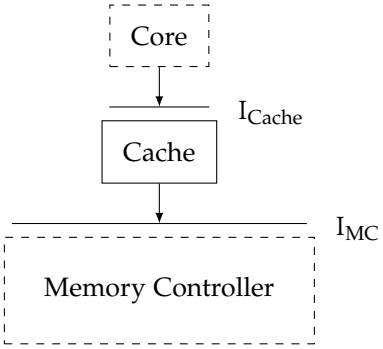


Figure 6.4: The cache in isolation

We can prove the memory controller is correct with respect to this abstract specification, meaning if the processor satisfies the requirements over users, then the results computed by the memory controller satisfy the requirements over results.

In such a case, the memory controller works as expected, but this does not mean that the result alone is sufficient to conclude about the system as a whole. We still have to prove the premise, *i.e.* that the processor makes a correct use of the memory controller interface. However, we do not need to use the model of the memory controller to that end, we only need to know that it will behave as specified by the abstract specification if the processor does too.

We focus on modeling the processor, made in our example of two components: a core and a cache. Similarly to what we achieved with the memory controller, we reason about the cache in isolation (as pictured with Figure 6.4). We want to prove that 1. the cache makes a correct use of the memory controller interface, *i.e.* each use of the memory controller interface satisfy the appropriate requirements, and 2. it provides similar guarantees to the core.

We proceed by defining a second abstract specification, this time against the interface of the cache. This additional abstract specification serves two purposes: it restricts how the core is allowed to use the cache, and by construction how the cache uses the memory controller, but it also formalizes the expectation of the core.

In this second case, the requirements over the interface results are the same as what we already introduced for the memory controller. Trying to prove a cache without SMRR can enforce them would uncover the SMRAM cache poisoning attack.

In this Section, we have discussed the limitations of MINIX86. This necessary assessment allowed us to propose an alternative formalism to address them. In the rest of this Chapter, we describe in depth this formalism.

6.2 Modeling Programs with Effects

The first objective of FreeSpec is to incrementally model a complex system, one component at a time. To do so, we use the key concepts of algebraic effects and effect handlers, implemented

with a variant of the Free monad called the Program monad as defined in the `operational` package of Haskell [130].

This section and the one afterwards proceed through a running example: a minimalist memory controller, as depicted in Figure 6.3. As before, we have implemented our proofs in Coq, and interesting readers can refer specifically to the file `examples/Smram.v` of [21].

6.2.1 Interface of Effects

Within a computing system, interconnected components communicate through interfaces. A component which exhibits an interface receives computational requests from other components; it handles these requests by computing their results and sending the latter back to the client component. In FreeSpec, a computational request is modeled with an effect, that is a symbolic value which describes the request and its potential result.

For $\mathcal{I}$ an interface, we denote by $\mathcal{I}|_{\mathcal{A}} \subseteq \mathcal{I}$ the subset of effects whose results belong to a set $\mathcal{A}$.

Example 6.4 (Memory Controller Interfaces)
The VGA and the DRAM controllers exhibit a similar interface which allows for reading and writing into a memory region. Their interfaces are denoted by $\mathcal{I}_{\text{VGA}}$ and $\mathcal{I}_{\text{DRAM}}$ respectively. Let $\mathcal{Loc}$ be the set of memory locations and $\mathcal{Val}$ the set of values stored inside the memory region. We use the unit value $()$ to model effects without results (similarly to the `void` keyword in an imperative language). We define $\mathcal{I}_{\text{DRAM}}$ (respectively $\mathcal{I}_{\text{VGA}}$) with two constructors:

$$\begin{aligned} \mathcal{I}_{\text{DRAM}} \quad &\triangleq \quad \mathit{Read}_{\text{DRAM}} : \mathcal{Loc} \to \mathcal{I}_{\text{DRAM}}|_{\mathcal{Val}} \\ &\mid \quad \mathit{Write}_{\text{DRAM}} : \mathcal{Loc} \to \mathcal{Val} \to \mathcal{I}_{\text{DRAM}}|_{\{()\}} \end{aligned}$$

Then, $\mathcal{I}_{\text{DRAM}} = \mathcal{I}_{\text{DRAM}}|_{\{()\}} \cup \mathcal{I}_{\text{DRAM}}|_{\mathcal{Val}}$, and $\mathit{Read}_{\text{DRAM}}(\mathtt{l}) \in \mathcal{I}_{\text{DRAM}}|_{\mathcal{Val}}$ is an effect that describes a memory access to read the value $\mathtt{v} \in \mathcal{Val}$ stored at the location $\mathtt{l} \in \mathcal{Loc}$.

The memory controller interface is similar, but it distinguishes between privileged and unprivileged accesses. It also provides one effect to lock the SMRAM protection mechanism, i.e. it enables the SMRAM isolation until the next hardware reset. We define the set $\mathcal{Priv} \triangleq \{\texttt{privileged}, \texttt{unprivileged}\}$ to distinguish between privileged memory accesses made by a processor in SMM and unprivileged accesses made the rest of the time. The memory controller interface, denoted by $\mathcal{I}_{\text{MCH}}$, is defined with three constructors:

$$\begin{aligned} \mathcal{I}_{\text{MCH}} \quad &\triangleq \quad \mathit{Read}_{\text{MCH}} : \mathcal{Loc} \to \mathcal{Priv} \to \mathcal{I}_{\text{MCH}}|_{\mathcal{Val}} \\ &\mid \quad \mathit{Write}_{\text{MCH}} : \mathcal{Loc} \to \mathcal{Val} \to \mathcal{Priv} \to \mathcal{I}_{\text{MCH}}|_{\{()\}} \\ &\mid \quad \mathit{Lock} : \mathcal{I}_{\text{MCH}}|_{\{()\}} \end{aligned}$$

6.2.2 Operational Semantics for Effects

An effect corresponds to a computational request made to an implementation of a given interface. To compute the result of the computational request, we define its *operational semantics*.

Ultimately, we will model a component as an operational semantics for all the effects of its interface. Since operational semantics are defined using a purely functional language, they always compute the same result for a given effect, which is inconsistent with the stateful aspect of hardware components. Thus, an operational semantics produces not only a result, but also a new operational semantics, which encapsulates the new state of the component.

Definition 6.1 (Operational Semantics)
We write $\Sigma_{\mathcal{I}}$ for the set of operational semantics for a given interface $\mathcal{I}$, defined co-inductively as

$$\Sigma_{\mathcal{I}} \triangleq \{\, \sigma \mid \sigma : \forall \mathcal{A}, \mathcal{I}|_{\mathcal{A}} \to \mathcal{A} \times \Sigma_{\mathcal{I}} \,\}.$$

An operational semantics $\sigma \in \Sigma_{\mathcal{I}}$ is a function which, given any effect of $\mathcal{I}$, produces both a result which belongs to the expected set and a new operational semantics to use afterwards.

A component may use more than one interface. For instance, the memory controller of our running example can access the system memory and the memory shared by the VGA controller. But an operational semantics is defined for only one interface. In FreeSpec, we solve this issue by composing interfaces together to create new ones.

Definition 6.2 (Interfaces Composition)
Let $\mathcal{I}$ and $\mathcal{J}$ be two interfaces. $\oplus$ is the interface composition operator, defined with two constructors:

$$\begin{aligned} \mathcal{I} \oplus \mathcal{J} \quad &\triangleq \quad \mathit{InL} : \forall \mathcal{A}, \mathcal{I}|_{\mathcal{A}} \to (\mathcal{I} \oplus \mathcal{J})|_{\mathcal{A}} \\ &\mid \quad \mathit{InR} : \forall \mathcal{A}, \mathcal{J}|_{\mathcal{A}} \to (\mathcal{I} \oplus \mathcal{J})|_{\mathcal{A}} \end{aligned}$$

The resulting interface $\mathcal{I} \oplus \mathcal{J}$ contains the effects of both $\mathcal{I}$ and $\mathcal{J}$, wrapped into either *InL* or *InR* constructors. Because constructors images are mutually exclusive [1], this means we can consider $\mathcal{I} \oplus \mathcal{I}$ that is the composition of $\mathcal{I}$ with itself. This is necessary if we want to be able to reason about a component connected to two other components which both exhibit the same interface. Besides, $\oplus$ preserves the effect result, *e.g.* given any effect $e \in \mathcal{I}|_{\mathcal{A}}$, then $\mathit{InL}(e) \in (\mathcal{I} \oplus \mathcal{J})|_{\mathcal{A}}$.

Example 6.5 (VGA and DRAM Composition)
We consider $\mathcal{I}_{\text{DRAM}} \oplus \mathcal{I}_{\text{VGA}}$. Then, $\mathit{InL}(\mathit{Read}_{\text{DRAM}}(\mathfrak{l})) \in (\mathcal{I}_{\text{DRAM}} \oplus \mathcal{I}_{\text{VGA}})|_{\mathcal{V}al}$ is an effect that describes a read access targeting the DRAM controller, whereas $\mathit{InR}(\mathit{Read}_{\text{VGA}}(\mathfrak{l})) \in (\mathcal{I}_{\text{DRAM}} \oplus \mathcal{I}_{\text{VGA}})|_{\{()\}}$ is an effect that describes a read access targeting the VGA controller.

Using $\oplus$, we can compose several interfaces together. We then need another composition operator, this time for operational semantics. We compose operational semantics together to construct a new operational semantics for the composed interface.

[1] See page xi

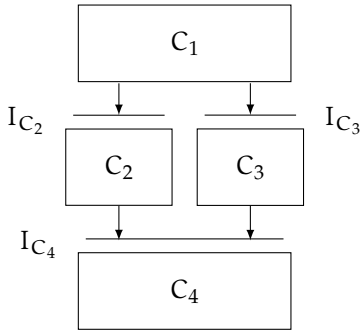


Figure 6.5: Illustration of the diamond pattern

Definition 6.3 (Operational Semantics Composition)
Let $\mathcal{I}$ and $\mathcal{J}$ be two interfaces, $\sigma_i \in \Sigma_{\mathcal{I}}$ and $\sigma_j \in \Sigma_{\mathcal{J}}$ be two operational semantics dedicated to these interfaces. We use the λ-calculus abstraction notation for unnamed functions. $\otimes$ is the composition operator for operational semantics, defined as

$$\sigma_i \otimes \sigma_j \triangleq \lambda e. \begin{cases} (x, \sigma_i' \otimes \sigma_j) & \text{when} \quad e = \mathit{InL}(e_i) \text{ and } \sigma_i(e_i) = (x, \sigma_i') \\ (x, \sigma_i \otimes \sigma_j') & \text{when} \quad e = \mathit{InR}(e_j) \text{ and } \sigma_j(e_j) = (x, \sigma_j') \end{cases}$$

The definition of $\otimes$ has an important impact over what can be specified in FreeSpec. Handling an effect of $\mathcal{I}$ (respectively $\mathcal{J}$) does not update σ_j (respectively σ_i). As a consequence, *we cannot specify as is a graph of components which contains a cycle or a diamond*. This is the main limitation of FreeSpec, but its incidence is abated because computing platforms are often designed as a hierarchical succession of layers. We illustrate it with the four-component system depicted in Figure 6.5. When C_1 uses an effect of the interface I_{C_2}, it is possible that the component C_2 will itself use an effect of the interface I_{C_4}. If that happens, the state of the component C_4, leading the result of the next effect used by C_3 to be different, compared it could have been without the action of C_2.

Using interfaces, we model how components interact with each other, and we aim to model a component which exposes an interface as an operational semantics for this interface. We now present a formalism –the Program monad— to model how a component uses its neighbors when it handles computational requests of its interface.

6.2.3 The Program Monad

Modeling programs with side effects in purely functional languages such as Gallina (the Coq specification language) or Haskell is usually achieved thanks to monads [131]. FreeSpec leverages a variant of the free monad called the Program monad [130] to model programs with effects. Operational semantics play the role of `operational` [130] interpreters. We write $P_{\mathcal{I}}(\mathcal{A})$ for the set of programs with effects which belongs to $\mathcal{I}$, modeled thanks to the Program monad,

and whose result belongs to a set $\mathcal{A}$.

Definition 6.4 (Program Monad)
The Program monad is defined with three constructors:

$$\begin{array}{rcl} \mathrm{P}_{\mathcal{I}}(\mathcal{A}) & \triangleq & \mathit{Pure} : \mathcal{A} \to \mathrm{P}_{\mathcal{I}}(\mathcal{A}) \\ & | & \mathit{Bind} : \forall \mathcal{B}, \mathrm{P}_{\mathcal{I}}(\mathcal{B}) \to (\mathcal{B} \to \mathrm{P}_{\mathcal{I}}(\mathcal{A})) \to \mathrm{P}_{\mathcal{I}}(\mathcal{A}) \\ & | & \mathit{Request} : \mathcal{I}|_{\mathcal{A}} \to \mathrm{P}_{\mathcal{I}}(\mathcal{A}) \end{array}$$

These constructors allow for the construction of values which act similarly to abstract syntax trees to model programs with effects. On the one hand, *Pure* and *Request* are comparable to the leaves of a syntax tree and model atomic computations; *Pure* models local computations, whereas *Request* models deferring a computational request to a handler and waiting for its result. On the other hand, *Bind* (usually written with the infix operator $\gg\!=$) models the control flow of a program with effects, like the abstract syntax tree nodes would. It defines how the result of one computation determines the following ones.

Example 6.6 (Copy)
We define $\mathit{copy} : \mathit{Loc} \to \mathit{Loc} \to \mathrm{P}_{\mathcal{I}_{\mathrm{DRAM}}}(\{()\})$ such that $\mathit{copy}(l, l')$ models a program with effects that returns no result, but copies the value v stored at the memory location l inside the memory location l'.

$$\mathit{copy}(l, l') \triangleq \mathit{Request}(\mathit{Read}_{\mathrm{DRAM}}(l)) \gg\!= \lambda v.\mathit{Request}(\mathit{Write}_{\mathrm{DRAM}}(l', v))$$

Given $l \in \mathit{Loc}$ and $l' \in \mathit{Loc}$, $\mathit{copy}(l, l')$ symbolically models a program with effects. To assign an interpretation of this program, it must be completed with an operational semantics which realizes the interface $\mathcal{I}_{\mathrm{DRAM}}$.

Definition 6.5 (Program With Effects Realization)
Let $\mathcal{I}$ be an interface, $\sigma \in \Sigma_{\mathcal{I}}$ an operational semantics for this interface and $\rho \in \mathrm{P}_{\mathcal{I}}(\mathcal{A})$ a program with effects which belong to this interface. $\sigma[\rho] \in \mathcal{A} \times \Sigma_{\mathcal{I}}$ denotes the realization of this program by σ, defined as:

$$\sigma[\rho] \triangleq \begin{cases} (x, \sigma) & \text{if} \quad \rho = \mathit{Pure}(x) \\ \sigma(e) & \text{if} \quad \rho = \mathit{Request}(e) \\ \sigma'[f(y)] & \text{if} \quad \rho = q \gg\!= f \text{ and } (y, \sigma') = \sigma[q] \end{cases}$$

6.2.4 Components as Programs with Effects

With the interfaces, their operational semantics, the $\oplus$ and $\otimes$ operators to compose them and the Program monad to model programs with effects which belong to these interfaces, we now have all we need to model a given component which exposes an interface $\mathcal{I}$ and uses another

interface $\mathcal{J}$. We proceed with the following steps: modeling the component in terms of programs with effects, then deriving one operational semantics for $\mathcal{I}$ from these programs, assuming an operational semantics for $\mathcal{J}$ has been provided.

The behavior of a component is often determined by a local, mutable state. When it computes the result of a computational request, not only a component may read its current state; but it can also modify it, for instance to handle the next computational request differently. This means we have to model the state of a component with a set $\mathcal{S}$ of symbolic state representations. We map the current state of the component and effects of $\mathcal{I}$ to a program with effects of $\mathcal{J}$. These programs must compute the effect result and the new state of the component.

Definition 6.6 (Component)
Let $\mathcal{I}$ be the interface exhibited by a component and $\mathcal{J}$ the interface it uses. Let $\mathcal{S}$ be the set of its states. The component C, defined in terms of programs with effects of $\mathcal{J}$, is of the form

$$\forall \mathcal{A}, \mathcal{I}|_{\mathcal{A}} \to \mathcal{S} \to \mathsf{P}_{\mathcal{J}}(\mathcal{A} \times \mathcal{S})$$

Hence, C specifies how the component handles computational requests, both in terms of computation results and state updates.

Example 6.7 (Minimal Memory Controller Model)
Let $\mathsf{C}_{\mathrm{MCH}}$ be the memory controller defined in terms of programs with effects of $\mathcal{I}_{\mathrm{DRAM}} \oplus \mathcal{I}_{\mathrm{VGA}}$, then $\mathsf{C}_{\mathrm{MCH}}$ is of the form

$$\forall \mathcal{A}, \mathcal{I}_{\mathrm{MCH}}|_{\mathcal{A}} \to \mathcal{S}_{\mathrm{MCH}} \to \mathsf{P}_{\mathcal{I}_{\mathrm{DRAM}} \oplus \mathcal{I}_{\mathrm{VGA}}}(\mathcal{A} \times \mathcal{S}_{\mathrm{MCH}})$$

where $\mathcal{S}_{\mathrm{MCH}} \triangleq \{\texttt{on}, \texttt{off}\}$ means the SMRAM protection is either activated (`on`) or deactivated (`off`).

On the one hand, the *Lock* effect will activate the isolation mechanism of the memory controller, setting its state to `on`. On the other hand, the effects constructed with $\mathit{Read}_{\mathrm{MCH}}$ and $\mathit{Write}_{\mathrm{MCH}}$ will use the current state of the memory controller, the privileged parameter of the effect and the memory location of the access to determine if it uses the DRAM or the VGA controller. By default, it fetches the memory of the DRAM controller, except if all of the three following conditions are satisfied : (1) the isolation mechanism is activated, (2) the access is unprivileged, and (3) the targeted memory location belongs to the SMRAM. In such a case, it reroutes access to the VGA controller.

A component C defined in terms of programs with effects cannot be used as is to compute the result of a given effect. To do that, we need to derive an operational semantics for $\mathcal{I}$ from C.

Definition 6.7 (Deriving Operational Semantics)
Let C be a component which exhibits an interface $\mathcal{I}$, uses an interface $\mathcal{J}$ and whose states belong to $\mathcal{S}$. Let $s \in \mathcal{S}$ be the current state of the component and $\sigma_{\mathrm{j}} \in \Sigma_{\mathcal{J}}$ be an operational semantics

for $\mathcal{J}$. We can derive an operational semantics for $\mathcal{I}$, denoted by $\langle C, s, \sigma_j \rangle$, defined as

$$\langle C, s, \sigma_j \rangle \triangleq \lambda i.(x, \langle C, s', \sigma'_j \rangle) \text{ where } ((x, s'), \sigma'_j) = \sigma_j[C(i, s)]$$

For each incoming effect $e \in \mathcal{I}$, we realize the program with effects associated with this effect, *i.e.* $C(e, s)$ where s is the current state of the component, with an operational semantics σ_j of $\mathcal{J}$. According the Definition 6.5, we get the result of this program with effect and a new operational semantics σ'_j. According to Definition 6.6, this result is a pair of the result of e and the new state s' of the component. Using s' and σ'_j, we can construct the new operational semantics of $\mathcal{I}$to use after handling e.

The resulting operational semantics models a system made of interconnected components, and can then be used to derive another component model into an operational semantics which models a larger system. For instance, we can proceed with the following steps to comprehensively model our running example: (i) defining the operational semantics for the DRAM and VGA controllers; (ii) using these operational semantics to derive an operational semantics from C_{MCH}. The resulting operational semantics can take part in the derivation of a cache defined in terms of programs with effects of $\mathcal{I}_{MCH}$, to model a larger part of the system pictured in the Figure 6.2.

6.3 Modular Verification of Programs with Effects

The first objective of FreeSpec is to provide the required tools to model each component of a system independently, and to compose these components to model the whole system. Its second objective is to verify that the composition of several components satisfies a set of properties. To achieve that, we introduce the so-called abstract specifications, which allows for specifying, for each interface, expected properties for the effect results, independently of any underlying handler. Abstract specifications can be used to emphasize the responsibility of each component of a system regarding the enforcement of a given security policy. Verifying a component is done against abstract specifications of the interfaces it directly uses, even if it relies on a security property enforced by a deeper component in the components graph. In this case, we have to verify that every single component which separates them preserve this property. This procedure can help to prevent or uncover architectural attacks.

In this section, we proceed with our running example by verifying that the memory controller correctly isolates the SMRAM. In order to do that, we define an abstract specification which states that privileged reads targeting the SMRAM returns the value which has previously been stored by a privileged write. It models the SMRAM isolation: unprivileged writes cannot tamper with the content of the SMRAM, as read by a privileged CPU.

6.3.1 Abstract Specification

In FreeSpec, an abstract specification dedicated to an interface $\mathcal{I}$ is twofold. It defines a precondition over the effects that a caller must satisfy; and, in return, it specifies a postcondition

over the effects results that an operational semantics must enforce. Since both the precondition and the postcondition may vary in time, we parameterize an abstraction specification with an abstract state and a step function to update this state after each effect realization.

Definition 6.8 (Abstract Specification)
An abstract specification A dedicated to an interface $\mathcal{I}$ is defined as a tuple $\langle \Omega, \mathit{step}, \mathit{pre}, \mathit{post} \rangle$ where

– Ω is a set of abstract states

– $\mathit{step} : \forall \mathcal{A}, \mathcal{I}|_{\mathcal{A}} \to \mathcal{A} \to \Omega \to \Omega$ is a transition function for the abstract state.

– *pre* is the precondition over effects, *i.e* a predicate on $\mathcal{I} \times \Omega$ such that $\mathit{pre}(e, \omega)$ is true if and only if the effect e satisfies the precondition parameterized with the abstract state ω.

– *post* is the postcondition over effects results, *i.e.* a predicate on $\bigcup_{\mathcal{A}} (\mathcal{I}|_{\mathcal{A}} \times \mathcal{A} \times \Omega)$ such that $\mathit{post}(e, x, \omega)$ is true if and only if the results x computed for the effects e satisfy the postcondition parameterized with the abstract state ω.

By defining an abstract specification of an interface $\mathcal{I}$, it becomes possible to abstract away the effect handler, i.e. the underlying component. As a consequence, reasoning about a program with effects can be achieved without the need to look at the effect handlers. An abstract specification is dedicated to one verification problem (in our context, one security property), and it is possible to define as many abstract specifications as required.

We write $\mathit{run}_{\mathit{step}} : \forall \mathcal{A}, \Sigma_{\mathcal{I}} \to \mathrm{P}_{\mathcal{I}}(\mathcal{A}) \to \Omega \to (\mathcal{A} \times \Sigma_{\mathcal{I}} \times \Omega)$ for the function which, in addition to realize a program with effects, updates an abstract state after each effect. Using $\mathit{run}_{\mathit{step}}$, we can determine both the precondition over effects and the postcondition over effects results while an operational semantics realizes a program with effects.

Example 6.8 (Memory Controller Abstract Specification)
Let A_{MCH} be the abstract specification such that $A_{\mathrm{MCH}} = \langle \Omega_{\mathrm{MCH}}, \mathit{step}_{\mathrm{MCH}}, \mathit{pre}_{\mathrm{MCH}}, \mathit{post}_{\mathrm{MCH}} \rangle$. A_{MCH} models the following property: "*privileged reads targeting the SMRAM return the value which has been previously stored by a privileged write*":

– Let $\mathit{Smram} \subseteq \mathit{Loc}$ be the set of memory locations which belong to the SMRAM. We define $\Omega_{\mathrm{MCH}} \triangleq \mathit{Smram} \to \mathit{Val}$, such that $\omega \in \Omega_{\mathrm{MCH}}$ models a view of the SMRAM as exposed by the MCH for privileged reads.

– We define $\mathit{step}_{\mathrm{MCH}}$ which updates the view of the MCH (modeled as a function) after each privileged write access targeting any SMRAM location l, that is

$$\mathit{step}_{\mathrm{MCH}}(e, x, \omega) \triangleq \begin{cases} \lambda l'.\ (\text{if } l = l' \text{ then } v \text{ else } \omega(l')) \\ \qquad \text{if } e = \mathit{Write}_{\mathrm{MCH}}(l, v, \texttt{privileged}) \text{ and } l \in \mathit{Smram} \\ \omega \qquad \text{otherwise} \end{cases}$$

– There is no precondition to the use of the memory controller effects, so

$$\forall e \in \mathcal{I}, \forall \omega \in \Omega_{\text{MCH}}, pre_{\text{MCH}}(e, \omega)$$

– The postcondition enforces that the result x of a privileged read targeting the SMRAM ($Read(l, \texttt{privileged})$) has to match the value stored in A_{MCH} abstract state, i.e. the expected content for this memory location $\omega(l)$.

$$post_{\text{MCH}}(e, x, \omega) \triangleq \forall l \in \mathcal{L}oc, e = Read_{\text{MCH}}(l, \texttt{privileged}) \wedge l \in \mathcal{S}mram \Rightarrow x = \omega(l)$$

6.3.2 Compliance and Correctness

The verification of a component C, which exhibits $\mathcal{I}$ and uses $\mathcal{J}$, consists in proving we can derive an operational semantics σ_i for $\mathcal{I}$ from an operational semantics σ_j for $\mathcal{J}$. This semantics σ_i enforces the postcondition of an abstract specification $A_{\mathcal{I}}$ dedicated to $\mathcal{I}$ (compliance). As C is defined in terms of programs with effects of $\mathcal{J}$, the latter needs to make a licit usage of $\mathcal{J}$ with respect to an abstract specification $A_{\mathcal{J}}$ dedicated to $\mathcal{J}$ (correctness).

First, σ_i complies with $A_{\mathcal{I}}$ if, (1) given any effect which satisfies $A_{\mathcal{I}}$ precondition, σ_i produces a result which satisfies its postcondition, and if (2) the new operational semantics σ_i' also complies with $A_{\mathcal{I}}$. The precondition and the postcondition are parameterized by an abstract state, so is the compliance property.

Definition 6.9 (Operational Semantics Compliance)
Let A be an abstract specification for an interface $\mathcal{I}$, defined as $\langle \Omega, step, pre, post \rangle$, $\omega \in \Omega$, then $\sigma \in \Sigma_{\mathcal{I}}$ complies with A in accordance with ω (denoted by $\sigma \models A[\omega]$) iff.

$$\forall e \in \mathcal{I}, pre(e, \omega) \Rightarrow post(e, x, \omega) \wedge \sigma' \models A[step(e, x, \omega)] \text{ where } (x, \sigma') = \sigma(e)$$

Secondly, programs with effects of C make a licit usage of an operational semantics $\sigma_j \in \Sigma_{\mathcal{J}}$ which complies with $A_{\mathcal{J}}$ if they only use effects which satisfy $A_{\mathcal{J}}$ precondition. As for the compliance property, correctness is parameterized with an abstract state.

Definition 6.10 (Program With Effects Correctness)
Let A be an abstract specification for an interface $\mathcal{I}$, defined as $\langle \Omega, step, pre, post \rangle$, $\omega \in \Omega$, and $\rho \in P_{\mathcal{I}}(\mathcal{A})$, then ρ is correct with respect to A in accordance with ω (denoted by $A[\omega] \dashv \rho$), iff.

$$A[\omega] \dashv \rho \triangleq \begin{cases} \text{True} & \text{if} \quad \rho = Pure(x) \\ pre(e, \omega) & \text{if} \quad \rho = Request(e) \\ \forall \sigma \in \Sigma_{\mathcal{J}} \text{ such that } \sigma \models A[\omega], & \\ \quad A[\omega] \dashv q \wedge A[\omega'] \dashv f(x) & \text{if} \quad \rho = q \gg\!= f \\ \text{where } (x, \_, \omega') = run_{step_{\mathcal{J}}}(\sigma, q, \omega) & \end{cases}$$

Every local computation (*Pure*) is correct with respect to A in accordance with ω. A computation which uses an effect $e \in \mathcal{I}$ (*Request*) is correct with respect to A in accordance with ω if and only if e satisfies the precondition of A for the abstract state ω. Finally, the chaining of two programs with effects (*Bind*) is correct with A in accordance with ω if the first program is correct with A in accordance with ω, and the second program is correct in accordance with the abstract state reached after the realization of the first program.

Properties, inferred from an abstract specification, of a correct program with effects only hold if it is realized by a compliant operational semantics. Besides, we prove that correct programs preserve operational semantics compliance.

Theorem 6.1 (Compliance Preservation)
Let A be an abstract specification dedicated to an interface $\mathcal{I}$, then σ a compliant operational semantics for $\mathcal{I}$ produces a compliant operational semantics σ' when it realizes a correct program ρ, that is

$$\sigma \models A[\omega] \wedge A[\omega] \dashv \rho \Rightarrow \sigma' \models A[\omega'] \text{ where } run_{step}(\sigma, \rho, \omega) = (x, \sigma', \omega')$$

As for interfaces (with $\oplus$) and operational semantics (with $\otimes$), we have also defined an abstract specification composition operator $\odot$. We do not detail its definition in this Chapter[2], but it has the significant property to allow for reasoning about the composition of interfaces and composition of operational semantics.

Theorem 6.2 (Congruent Composition)
Let $\mathcal{I}$ (respectively $\mathcal{J}$) be an interface. Let $A_{\mathcal{I}}$ (respectively $A_{\mathcal{J}}$) be an abstract specification and $\sigma_i \in \Sigma_{\mathcal{I}}$ (respectively $\sigma_j \in \Sigma_{\mathcal{J}}$) be an operational semantics for this interface.

$$\sigma_i \models A_{\mathcal{I}}[\omega_i] \wedge \sigma_j \models A_{\mathcal{J}}[\omega_j] \Rightarrow \sigma_i \otimes \sigma_j \models (A_{\mathcal{I}} \odot A_{\mathcal{J}})[\omega_i, \omega_j]$$

With the compliance preservation, we know that as long as we follow the abstract specification precondition related to the effects we use, compliant operational semantics keep enforcing the postcondition. With the compliance preservation and congruent composition, we know we can reason locally, that is component by component.

6.3.3 Proofs Techniques to Show Compliance for Components

We have dived into the mechanisms which allow for composing together compliant operational semantics, but little has been said about how to prove the compliance property. In a typical FreeSpec use case, operational semantics are not built as is, but rather derived from a component model (Definition 6.7). How to prove the resulting operational semantics complies with an abstract specification depends on how the component is connected to the rest of the system. We

[2]Interesting readers can refer to the file `theories/Compose.v` of [21] to find its definition, but the latter is not required to understand the rest of this Chapter.

have already discussed the consequences of the operational semantics composition operator $\otimes$ (Definition 6.3). Notably, a graph of components which contains a cycle, a diamond or a forward edge cannot be easily modeled and verified in FreeSpec. In its current state, FreeSpec provides two theorems to verify the properties of a component model in terms of an abstract specification, dedicated to two composition patterns: when a component is being used by one component, and when a component is being used by more than one component.

Predicate of Synchronization. The simplest scenario consists of one component which uses many components, and is only used by one other component, *e.g.* in Figure 6.3. Let $\mathcal{I}$ and $\mathcal{J}$ be two interfaces and let C be a component with a set of states $\mathcal{S}$, which exhibits $\mathcal{I}$ and uses $\mathcal{J}$. Let $A_{\mathcal{I}}$ be an abstract specification dedicated to $\mathcal{I}$. Deriving an operational semantics from C which complies with $A_{\mathcal{I}}$ in accordance with $\omega_i \in \Omega_I$ requires to show the existence of $s \in \mathcal{S}$ and $\sigma_j \in \Sigma_{\mathcal{J}}$ such that

$$\langle C, s, \sigma_j \rangle \models A_{\mathcal{I}}[\omega_i].$$

However, proving this statement would not be very satisfying, as it ties our verification results to one specific operational semantics σ_j, and by extension one specific component. As a consequence, we define an abstract specification $A_{\mathcal{J}}$ to generalize our statement and abstracting away σ_j. We now need to prove there exists $\omega_j \in \Omega_{\mathcal{J}}$ such that given an operational semantics σ_j which complies with $A_{\mathcal{J}}$ in accordance with ω_j, the operational semantics derived from C, s and σ_j complies with $A_{\mathcal{I}}$ in accordance with ω_i, that is

$$\forall \sigma_j \in \Sigma_{\mathcal{J}},\ \sigma_j \models A_{\mathcal{J}}[\omega_j] \Rightarrow \langle C, s, \sigma_j \rangle \models A_{\mathcal{I}}[\omega_i]$$

The combinatorial explosion of cases introduced by ω_i, s and ω_j, modified as the component handles effects, makes inductive reasoning challenging. The FreeSpec framework provides a useful theorem to address these challenges, which leverages a so-called predicate of synchronization. The latter is defined by the user on a case-by-case basis, to act as an invariant for the induction, and a sufficient condition to enforce compliance.

Theorem 6.3 (Derivation Compliance)
Let *sync*, a relation between abstract states of $\Omega_{\mathcal{I}}$ and $\Omega_{\mathcal{J}}$ and concrete states of $\mathcal{S}$, be a predicate of synchronization. Then, it is expected that, $\forall \omega_i \in \Omega_{\mathcal{I}}$, $s \in \mathcal{S}$ and $\omega_j \in \Omega_{\mathcal{J}}$ such that $sync(\omega_i, s, \omega_j)$ holds, then $\forall \sigma_j \in \Sigma_{\mathcal{J}}$ such that $\sigma_j \models A_{\mathcal{J}}[\omega_j]$ and $\forall e \in \mathcal{I}$ such that $pre_{\mathcal{I}}(e, \omega_i)$,

1. C preserves the synchronization of states, that is $sync(\omega_i', s', \omega_j')$
2. C is defined in terms of programs with effects which are correct with respect to $A_{\mathcal{J}}$ in accordance with ω_j, that is $A_{\mathcal{J}}[\omega_j] \dashv C(e, s)$
3. C computes a result for e which satisfies $A_{\mathcal{I}}$ postcondition, that is $post_{\mathcal{I}}(e, x, \omega_i)$

where $((x, s'), \sigma_j', \omega_j') = run_{step_{\mathcal{J}}}(\sigma_j, C(e, s), \omega_j)$ and $\omega_i' = step_{\mathcal{I}}(e, x, \omega_i)$.

Should these three properties be verified, then we show that

$$sync(\omega_i, s, \omega_j) \wedge \sigma_j \models A_{\mathcal{J}}[\omega_j] \Rightarrow \langle C, s, \sigma_j \rangle \models A_{\mathcal{J}}[\omega_i].$$

Example 6.9 (Memory Controller Compliance)
We want to prove we can derive an operational semantics from C_{MCH} (Example 6.7) which complies with A_{MCH} (Example 6.8).

We define $A_{DRAM} \triangleq \langle \Omega_{DRAM}, step_{DRAM}, pre_{DRAM}, post_{DRAM} \rangle$ an abstract specification dedicated to $\mathcal{I}_{DRAM}$ to express the following property: *"a read access to a memory location which belongs to the SMRAM return the value which has been previously written at this memory location."* In particular, $\Omega_{DRAM} = \Omega_{MCH}$, i.e. they are two views of the SMRAM, as exposed by the DRAM controller or by the memory controller. In this context, the behavior of VGA is not relevant. Let $\top$ be the abstract specification which has no state and such that its precondition and postcondition are always satisfied (meaning every operational semantics always complies with it). Therefore, the abstract specifications dedicated to the interface used by C_{MCH}, that is $\mathcal{I}_{DRAM} \oplus \mathcal{I}_{VGA}$, is $A_{DRAM} \odot \top$ whose abstract state is Ω_{DRAM}.

We define the predicate of synchronization $sync_{MCH}$ such that

$$sync_{MCH}(\omega_i, s, \omega_j) \triangleq s = \mathtt{on} \wedge \forall l \in \mathcal{S}ram, \omega_i(l) = \omega_j(l)$$

Hence, we start our reasoning from a situation where the SMRAM isolation is already activated and the states of the two abstract specifications are the same, meaning the two views of the SMRAM (as stored in the DRAM, and as exposed by the memory controller) coincide. We prove $sync_{MCH}$ satisfies the three premises of the Theorem 6.3. We conclude we can derive an operational semantics from C_{MCH} which complies with A_{MCH}.

Predicate of Neutrality. Another common composition pattern consists of a component which is used by more than one other component, *e.g.* the memory controller in Figure 6.2 is being used by the IOMMU and the cache. FreeSpec provides a theorem which allows for extending the result obtained with the Theorem 6.3, in the specific case where concurrent accesses do not lead to any abstract state update, and always satisfy the requirements over effects. This represents an important constraint, but matches realistic use cases. In particular, when hardware components interfaces are very large, two components are likely to use different classes of effects that do not interfere with each other.

Definition 6.11 (Predicate of Neutrality)
Let $\mathcal{I}$ be an interface and $A \triangleq \langle \Omega, step, \mathbb{P}, \mathbb{Q} \rangle$ be an abstract specification dedicated to $\mathcal{I}$. Let $pn : \mathcal{I} \to \mathbb{P}$ be a subset of effects. pn is said to be a predicate of neutrality of A (denoted $A \| pn$)

if and only if

$$\forall \mathcal{A},\ \omega \in \Omega,\ i \in \mathcal{I}|_{\mathcal{A}},\ x \in \mathcal{A},$$
$$(pn(i) \Rightarrow \mathbb{P}(i,\omega)) \wedge (pn(i) \wedge \mathbb{Q}(i,x,\omega) \Rightarrow \omega = step(i,x,\omega))$$

In a similar manner to the predicate of synchronization, FreeSpec provides a theorem to help and guide the verification work of its users.

Theorem 6.4 (Neutrality and Concurrency)
Let $\sigma \in \Sigma_{\mathcal{I}}$ be an operational semantics dedicated to $\mathcal{I}$. Let $pn : \mathcal{I} \to \mathbb{P}$ be a subset of effects. Let $\sigma\downarrow pn$ be the operational semantics built upon σ which executes arbitrary sequences of effects satisfying pn between two effects. We prove that

$$A \| pn \wedge \sigma \models A[\omega] \Rightarrow \sigma\downarrow pn \models A[\omega]$$

In other words, a second component which only uses the effects that satisfies pn can use the interface concurrently to a component proved to be correct with respect to A.

Example 6.10 (Memory Controller and IOMMU)
As pictured in Figure 6.2, the memory controller not only arbitrates the memory accesses of the CPU, but also to other hardware components (mostly PCI and PCIe devices). One possible predicate of neutrality for A_{MCH} is to deny the IOMMU the possibility to perform privileged write, so that only the main CPU could do it, that is

$$pn_{\text{IOMMU}}(e) \triangleq \forall l \in \mathcal{L}oc, v \in \times \mathcal{V}al, e \neq Write_{\text{MCH}}(l, v, \texttt{privileged})$$

By definition of A_{MCH}, and more precisely according to its transition function, only privileged write accesses update its abstract state. Therefore, we prove

$$\forall \mathcal{A}, \omega \in \Omega_{\text{MCH}}, e \in \mathcal{I}|_{\mathcal{A}}, x \in \mathcal{A}, pn_{\text{IOMMU}}(e) \Rightarrow step_{\text{MCH}}(\omega, e, x, \omega) = \omega$$

Besides, the precondition pre_{MCH} always holds true.

As a consequence, $A_{\text{MCH}} \| pn_{\text{IOMMU}}$, that is pn_{IOMMU} is a predicate of neutrality of A_{MCH}. This means if we can prove that the effects used by the IOMMU always satisfy pn_{IOMMU} predicate, then we can safely compose it with a cache knowing that, from the security perspective of the cache, it is like the IOMMU "is not here" (it does not interfere).

6.4 Conclusion

In this Chapter, we have introduced the FreeSpec key definitions and theorems so that we could model a minimal memory controller component and verify its properties in the presence of a well-behaving DRAM controller. This example has been driven by a real mechanism commonly found inside x86-based computing platforms.

The typical workflow of FreeSpec can be summarized as follows: specifying the interfaces of a system; modeling the components of the system in terms of programs with effects of these interfaces; identifying the abstract specifications which express the requirements over each interface; verifying each component in terms of compliance with these abstract specifications.

Independent groups of people can use FreeSpec to modularly model and verify a system, as long as they agree on the interfaces and abstract specifications. If, during the verification process, one group finds out a given interface or abstract specification needs to be updated, the required modifications may impact its neighbors. In other words, this verification process can help uncover inconsistencies in hardware specifications which pave the road towards compositional attacks. For instance, modeling a x86-based computing system, as pictured in Figure 6.2, using FreeSpec requires to take into account the cache, and to verify it complies with an abstract specification similar to the one defined in Example 6.8. As we explained in Example 6.2, such a proof was not possible to write prior to 2009, because the cache was lacking an access control mechanism for the SMRAM at that time. Thus, FreeSpec could have helped uncover the SMRAM cache poisoning attack previously mentioned [5, 4].

The abstract specifications are defined in terms of interfaces, *i.e.* independently from components. It has two advantages. First, for a given verification problem modeled with a set of abstract specifications, two components which exhibit the same interface can be proven to comply with the same abstract specification. In such a case, we can freely interchange these components, and the verification results remain true. This is useful to consider the challenge posed by components versioning, i.e. a new version of a component brings new features which could be leveraged by an attacker. Then, it is possible to verify a given component in terms of several abstract specifications. This means we can independently conduct several verification works against the same component.

7

Conclusion and Perspectives

" I suppose it is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail."

— Abraham Maslow

The starting point of this thesis is the SMRAM cache poisoning attack [4, 5], and more generally various compositional attacks [3] against the x86 hardware architecture [1, 6, 7]. These attacks have in common to leverage inconsistencies in the hardware specifications to defeat security policies supposedly enforced by the hardware architecture.

In this thesis, we have identified a class of security enforcement mechanisms called Hardware-based Security Enforcement (HSE) mechanisms, which consist of the configuration by a trusted software component of the underlying hardware architecture in order to constrain the execution of untrusted software components with respect to a targeted security policy. We have investigated the use of formal methods to formally specify and verify HSE mechanisms as a potential solution to uncover compositional attacks. We steer a middle course between two domains: hardware verification and system software verification. Generally, hardware verification focuses on properties which are transparent to the executed software and system software verification relies on hardware models which abstract the architecture complexity as much as possible . On the contrary, when it comes to HSE mechanisms it is important to consider that (1) hardware architectures often allow for implementing several HSE mechanisms, and (2) hardware features involved in HSE mechanisms are not safe by default.

In the following, we first give a brief summary of our two contributions and then suggest some possible directions for future work.

7.1 Summary of the Contributions

Our contribution is twofold. As a first step, we have proposed a theory of HSE mechanisms. Furthermore, we have proposed a compositional reasoning framework for Coq, based on our experience in implementing a proof of concept for our theory.

A Theory of HSE Mechanisms. We have proposed in Chapter 4 a theory of HSE mechanisms, such that a mechanism is primarily characterized by a set of trusted software components, a set of requirements over states and a set of requirements over software transitions. We have evaluated our approach in Chapter 5. We have introduced MINX86, a minimal model for a single core x86-based computing platform, and we have used it to specify and verify the HSE mechanism implemented by the BIOS to remain isolated from the rest of the software stack at runtime. We have written machine-checked proofs in Coq to increase our confidence in our result. Proofs related to the Chapter 4 have been commented in Appendix A and those related to the Chapter 5 have been released as free software [37].

Compositional Reasoning for Coq. We have proposed in Chapter 6 a novel approach which enables modular verification of complex systems made of interconnected components. This approach is the result of various lessons learned during the development of our first proof of concept, and we believe it represents a first step towards addressing the challenge posed by the scale of the x86 hardware architecture. Components of a system are primarily identified by the interface they expose, and secondarily by their current state and the interfaces they use. We have introduced so-called abstract specifications to allow for reasoning about components in isolation *and* about the expected properties of their composition. Besides, the resulting Coq framework, called FreeSpec and also made available as free software [21], is not specific to hardware models, and could also be leveraged to reason about composition of software components as well.

7.2 Perspectives

This thesis introduces a theory for specifying and verifying HSE mechanism as restrictions on hardware models, and a formalism to define and reason about these models in terms of interconnected components.

Using FreeSpec to Specify and Verifying HSE Mechanisms

Our theory of HSE mechanisms and our compositional reasoning framework for Coq remain two separated projects. The most natural continuation of our work would be to connect them, *e.g.* by substituting MINX86 with a model developed using FreeSpec. Going further, we are convinced a general-purpose model for the x86 architecture would be worth the time and effort spent on its construction. However, we have no doubt that the limitations of FreeSpec —in

terms of components interaction patterns— still reduce its applicability for the most complex parts of the x86 hardware architecture.

Extending FreeSpec With a Model Validation Framework

As a complement, the trustworthiness of a general-purpose hardware model is of key importance, as emphasized by Reid *et al.* for their formal specification of ARM [107]. Extending FreeSpec with a validation model framework would be an important step in that direction. We have taken great care for FreeSpec to be compatible with the code extraction feature of Coq, which means we can turn components model into executable programs. This feature opens interesting opportunities, but remains only a first step towards a practical model validation framework. In addition to traditional challenges, validating a model of a PCH, for instance, promises to be challenging because of its tight integration inside Intel chips. Focusing our efforts on open source processors, such as Leon4 [132], could allow us to investigate further this avenue of research.

A

A Formal Definition of HSE Mechanisms in Coq

This Appendix presents an implementation of the formal definition of HSE mechanisms detailed in Chapter 4, and follows a similar outline. The main purpose of this development is to provide rigorous, machine-checked proofs of the lemmas and theorems discussed in the Chapter. We assume the reader is familiar with Coq, and we discuss several key fragments of the development.

A.1 Hardware Model

A.1.1 Definition

A hardware model in our formalism is a tuple $\langle H, L_S, L_H, \rightarrow \rangle$ (Definition 4.1), with $\rightarrow$ being a predicate on $H \times (L_S \uplus L_H) \times H$.

The three sets H, L_S and L_H are introduced as variables of our development. This means they are implicit arguments of any further definition which uses them.

```
Variables (H Ls Lh:  Type).
```

The disjoint union $\uplus$ is modeled through a dedicated inductive type called `label`.

```
Inductive label :=
| Software
  : Ls -> label
| Hardware
  : Lh -> label.
```

Because H, L_S and L_H are *implicit* arguments of our development, a hardware model can be reduced to its relation transition $\rightarrow$.

```
10 Definition model
11   : Type :=
12   H -> label -> H -> Prop.
```

A transition in our formalism is a tuple (h, l, h') which satisfies the transition relation of the model. Subsets in Coq are usually modeled using so-called sigma-type: `{ x: A | P x }` is the subset of elements of type `A` which satisfy the predicate `P`. We define `transition m`, the set of transitions of a model `m`, using a sigma-type.

```
29 Definition transition
30            (m:  model)
31   : Type :=
32   { tr | m (from tr) (labelled tr) (to tr) }.
```

Because Gallina is a strongly typed language, manipulating a sigma-type can be cumbersome. In particular, there is no implicit coercion from `{ x: A | P x }` to `A` by default. The function `proj1_sig` can be used to explicitly coerce a sigma-type value, and we leverage the `Notation` feature of Coq, in order to ease the coercion.

```
34 Notation "'#' x" :=
35   (proj1_sig x) (at level 0).
```

That is, when we write `#x`, Coq will unwrap the sigma-type.

A.1.2 Traces

The next step is to model traces (Definition 4.2). We first introduce `sequence`, a parameterized type which cannot be empty. This simplifies several definitions, such as `init` (which returns the initial state of a trace).

```
39 Inductive sequence
40           (a:  Type)
41   : Type :=
42 | Ssingleton (x:  a)
43   : sequence a
44 | Scons (x:    a)
45         (rst:  sequence a)
46   : sequence a.
```

Not all sequences are traces, as a trace is a sequence where, given two consecutive transitions, the initial state of the second one is the resulting state of the first. Similarly to the `transition` type, we define `trace` with a sigma-type. To define the predicate to distinguish between valid and invalid trace, we first define `init` and `trace` as functions on `sequence (transition m)`, with the set of transitions returned by `trace` is modeled as a predicate on `transition m`. Then, we define `is_trace`, an inductive predicate on `sequence (transition m)`.

```
Inductive is_trace
          {m:  model}
  : sequence (transition m) -> Prop :=
| singleton_is_trace (tr:  transition m)
  : is_trace (Ssingleton tr)
| step_is_trace (tr:   transition m)
                (rho:  sequence (transition m))
                (Heq:  to #tr = init rho)
                (Hrec:  is_trace rho)
  : is_trace (Scons tr rho).
```

Finally, we use the `is_trace` predicate to define `trace m`.

```
Definition trace
           (m:  model)
  : Type :=
  { tr:  sequence (transition m) | is_trace tr }.
```

A.1.3 Security Policies

We have detailed how security policies can be modeled in transition systems, in Subsection 3.1.2 for the general case and in Subsection 4.1.3 for the context of HSE mechanisms. A security policy is either a predicate on sets of traces, a predicate on traces or a predicate on transitions. In this development, we keep the former (predicate on sets of traces) as the generic definition.

```
Definition security_policy
           (m:  model) :=
  (trace m -> Prop) -> Prop.
```

We then express the two latter (predicate on traces, and predicate on transitions) as particular sub-cases of this generic definition.

```
Definition security_property
           (m:  model)
           (prop:  trace m -> Prop)
  : security_policy m :=
  fun (traces:  trace m -> Prop)
  => forall (rho:  trace m), traces rho -> prop rho.
```

```
Definition safety_property
           {m:     model}
           (prop:  transition m -> Prop)
  : security_policy m :=
  fun (traces:  trace m -> Prop)
```

```
106   => forall (rho:  trace m),
107       traces rho
108       -> forall (tr: transition m),
109         trans #rho tr
110         -> prop tr.
```

A.2 HSE Mechanisms

We now give a formal definition of HSE mechanisms in the Coq theorem prover, as stated in Definition 4.4.

A.2.1 Definition and HSE Laws

First, we introduce an helper definition to easily express predicates of the form $l \in L_S \Rightarrow P(l)$.

```
112 Definition if_software
113             (l:  label)
114             (P:  Ls -> Prop)
115   : Prop :=
116   match l with
117   | Software l
118     => P l
119   | _
120     => True
121   end.
```

The `if_software` allows for hiding the pattern matching that is necessary to express various definitions of our theory of HSE mechanisms. Then, we define a type of HSE mechanisms using the Coq `Record` syntax, as follows.

```
123 Record HSE
124        (m:  model) :=
125   { software:      Type
126   ; tcb:           software -> Prop
127   ; context:       H -> software
128   ; hardware_req:  H -> Prop
129   ; software_req:  H -> Ls -> Prop
130   ; law_2:         forall (tr:  transition m),
131       hardware_req (from #tr)
132       -> if_software (labelled #tr)
133                      (software_req (from #tr))
134       -> hardware_req (to #tr)
```

```
135   ; law_1:         forall (tr:  transition m),
136       ~ tcb (context (from #tr))
137       -> if_software (labelled #tr)
138                      (software_req (from #tr))
139   }.
```

By making the two laws part of the HSE mechanisms definition, we ensure that no inconsistent HSE mechanism can be defined in Coq.

A.2.2 Trace Compliance

Whether trusted software components are correctly implementing a given HSE mechanism is a safety property. This means we shall be able to derive a subset of "compliant" traces from the set of traces of the hardware model.

```
147   Definition compliant_trace
148              {m:    model}
149              (hse:  HSE m)
150              (rho:   trace m)
151     : Prop :=
152     hardware_req hse (init #rho)
153     /\ forall (tr:  transition m),
154       trans #rho tr
155       -> if_software (labelled #tr)
156                      (software_req hse (from #tr)).
```

This definition is straightforward: for a trace to be compliant with a HSE mechanism, its initial state has to comply with the hardware requirements of this HSE mechanism, while its software transitions (that is, transitions whose label belongs to `Ls`) satisfy its software requirements.

In order to prove Lemma 4.1, we first prove an intermediary result: if a trace complies with a given HSE mechanism, then the subtrace obtained by removing its first transition also complies with the HSE mechanism.

```
158   Fact compliant_trace_rec
159        {m:      model}
160        (hse:    HSE m)
161        (x:      transition m)
162        (rho:    sequence (transition m))
163        (Hcons:  is_trace (Scons x rho))
164        (Hrho:   is_trace rho)
165     : compliant_trace hse (exist (Scons x rho) Hcons)
166       -> compliant_trace hse (exist rho Hrho).
```

```
Proof.
  intros [Hhard_req Hsoftware_req].
  constructor.
  + inversion Hcons; subst.
    cbn.
    rewrite <- Heq.
    apply law_2.
    ++ apply Hhard_req.
    ++ apply Hsoftware_req.
       left; reflexivity.
  + intros tr Htrans.
    apply Hsoftware_req.
    right; exact Htrans.
Qed.
```

We use this result in the proof by induction of Lemma 4.1. This lemma states that hardware requirements of a consistent HSE mechanism are invariant of compliant traces.

```
Lemma hse_inv_enforcement
      {m:    model}
      (hse:  HSE m)
  : forall (rho:  trace m),
    compliant_trace hse rho
    -> forall (tr:  transition m),
      trans #rho tr
      -> hardware_req hse (from #tr)
         /\ hardware_req hse (to #tr).
Proof.
  intros [rho Hrho] Hcomp tr Htrans.
  cbn in *.
  induction rho.
```

The first case to consider is the singleton sequence, with only one transition. By definition of the compliant traces, the initial state of the only transition of `rho` satisfies `hardware_req`. In order to prove that the resulting state of this transition also satisfies the requirement, we use the second law of the HSE mechanism definition.

```
  + inversion Hcomp as [Hhard_req Hsoftware_req].
    split.
    ++ now rewrite Htrans.
    ++ apply law_2.
       +++ now rewrite Htrans.
       +++ now apply Hsoftware_req.
```

The second case to consider is a trace made of an initial transition and a subtrace. If we consider the associated set of transitions, we again have to cover two cases. Firstly, we can focus on the initial transition: the proof is here very similar to the singleton trace case. Secondly, we can consider the transitions of the subtrace. We know this subtrace is compliant using `compliant_trace_rec`. This allows us to use the induction hypothesis, and conclude the proof.

```
  + destruct Htrans as [Htrans|Htrans].
    ++ rewrite Htrans.
       split; [apply Hcomp |].
       apply law_2.
       +++ apply Hcomp.
       +++ apply Hcomp.
           left; reflexivity.
    ++ inversion Hrho; subst.
       apply (IHrho Hrec).
       +++ eapply compliant_trace_rec.
           apply Hcomp.
       +++ apply Htrans.
Qed.
```

A.2.3 HSE Mechanism Correctness

```
Definition correct_hse
           {m:    model}
           (hse:  HSE m)
           (p:    security_policy m)
  : Prop :=
  p (compliant_trace hse) .
```

```
Theorem safety_property_correct_hse
        {m:    model}
        (hse:  HSE m)
        (p:    transition m -> Prop)
  : (forall (rho:  trace m)
            (tr:   transition m),
        trans #rho tr
        -> hardware_req hse (from #tr)
        -> if_software (labelled #tr)
                       (software_req hse (from #tr))
        -> p tr)
    -> correct_hse hse (safety_property p).
Proof.
```

```
251   intros Hreq.
252   unfold correct_hse, safety_property.
253   intros rho Hcomp tr Htrans.
254   apply (Hreq rho tr Htrans).
255   + eapply hse_inv_enforcement.
256     ++ exact Hcomp.
257     ++ exact Htrans.
258   + apply Hcomp.
259     exact Htrans.
260 Qed.
```

A.2.4 HSE Mechanisms Composition

HSE mechanisms are commonly implemented simultaneously by modern software stack, *e.g.* an operating system configures the MMU, while firmware relies on the processor SMM. In Chapter 4, we have proposed a first definition of a composition operator to reason about such scenario. This operator is not total, and cannot compose together two arbitrary HSE mechanisms. Therefore, we first define a predicate to determine whether two HSE mechanisms are compatible or not. The main characteristic of two compatible HSE mechanisms is that they use the same hardware-software mapping. Maybe surprisingly for the reader unfamiliar with Coq, this property is not straightforward to express and requires to use the `eq_rect` function which allows to cast a value from one type to another, assuming we can provide the proof both types are actually the same.

```
401 Definition compatible_hse
402            {m:  model}
403            (hse1 hse2:  HSE m)
404   : Prop :=
405   software hse1 = software hse2
406   /\ forall x H,
407     context hse2 x = eq_rect _ id (context hse1 x) _ H.
```

We then define the `HSE_cap` function (`cap` being the name of the latex command of $\cap$) which computes the composition of two compatible HSE mechanisms. We rely on the `refine` tactic in order to write the proofs of consistency of the resulting HSE mechanisms in interactive mode (using tactics) rather than providing them directly.

```
409 Definition HSE_cap
410            {m:            model}
411            (hse1 hse2:    HSE m)
412            (Hcompatible:  compatible_hse hse1 hse2)
413   : HSE m.
414   refine ({| software := software hse1
```

```
415             ; context  := context hse1
416             ; tcb := fun x
417                     => tcb hse1 x \/ tcb hse2 (eq_rect _ id x _ _)
418             ; hardware_req := fun x
419                               => hardware_req hse1 x
420                                  /\ hardware_req hse2 x
421             ; software_req := fun x l
422                              => software_req hse1 x l /\ software_req hse2 x l
423             |}).
```

As stated in Definition 4.5, we prove the first law...

```
429   + intros tr Htcb.
430     apply Classical_Prop.not_or_and in Htcb.
431     destruct Htcb as [Ht1 Ht2].
432     inversion Hcompatible as [Hsoftware Hcontext].
433     rewrite <- Hcontext in Ht2.
434     apply law_1 in Ht1.
435     apply law_1 in Ht2.
436     induction tr as [tr Htr];
437       induction tr as [[h l] h'];
438       induction l; auto.
439     unfold proj1_sig.
440     split.
441     ++ apply Ht1.
442     ++ apply Ht2.
443     Unshelve.
444     apply Hcompatible.
445 Defined.
```

... and the second law (`hardware_req` is an invariant).

```
424   + intros tr [Hi1 Hi2] Hsoft.
425     split; apply law_2; [exact Hi1 | idtac | exact Hi2 | idtac];
426        induction tr as [tr Htr]; induction tr as [[h l] h'];
427        induction l; auto;
428        apply Hsoft.
```

We discussed in Chapter 4 several expected properties of the HSE mechanisms composition. For instance, Lemma 4.3 states that the set of compliant traces of the compositions of two HSE mechanisms is the intersection of the sets of compliant traces of each HSE mechanism. In the context of this development, sets are defined as predicates (this is a common approach in Coq). To reason about set equality, we prove two complementary implications. Firstly, a compliant

trace of the composition of two HSE mechanisms necessarily complies with one of these HSE mechanisms. The associated proof covers both possible cases whose are very similar [1].

```
Lemma compliant_trace_intersec_intersec_compliant_trace
      {m:      model}
      (hse_1:  HSE m)
      (hse_2:  HSE m)
      (Hcomp:  compatible_hse hse_1 hse_2)
      (rho:    trace m)
  : compliant_trace (HSE_cap hse_1 hse_2 Hcomp) rho
    -> compliant_trace hse_1 rho /\ compliant_trace hse_2 rho.
Proof.
  intros Hct.
  unfold compliant_trace in Hct.
  unfold HSE_cap in Hct.
  cbn in Hct.
  split.
  + constructor.
    ++ apply Hct.
    ++ intros tr Htr.
       destruct Hct as [_H Hct].
       assert (Hres:  if_software (labelled #tr)
                                  (fun l : Ls =>
                                     software_req hse_1 (from #tr) l
                                     /\ software_req hse_2 (from #tr) l))
         by (apply Hct; exact Htr).
       induction tr as [tr _H2]; induction tr as [[h l] h'].
       induction l; auto.
       cbn in *.
       apply Hres.
  + constructor.
    ++ apply Hct.
    ++ intros tr Htr.
       destruct Hct as [_H Hct].
       assert (Hres:  if_software (labelled #tr)
                                  (fun l : Ls =>
                                     software_req hse_1 (from #tr) l
                                     /\ software_req hse_2 (from #tr) l))
         by (apply Hct; exact Htr).
       induction tr as [tr _H2]; induction tr as [[h l] h'].
```

[1] We can probably reduce the size of the proof by half with some refactoring.

```
509         induction l; auto.
510         cbn in *.
511         apply Hres.
512   Qed.
```

Secondly, a trace which complies with one of two HSE mechanisms also complies with their composition.

```
514   Lemma intersec_compliant_trace_compliant_trace_intersec
515         {m:      model}
516         (hse_1:  HSE m)
517         (hse_2:  HSE m)
518         (Hcomp:  compatible_hse hse_1 hse_2)
519         (rho:    trace m)
520     : compliant_trace hse_1 rho /\ compliant_trace hse_2 rho
521       -> compliant_trace (HSE_cap hse_1 hse_2 Hcomp) rho.
522   Proof.
523     intros [H1 H2].
524     constructor.
525     + unfold HSE_cap.
526       cbn.
527       split; [apply H1|apply H2].
528     + intros tr Htrans.
529       assert (Hb1:  if_software (labelled #tr)
530                                 (software_req hse_1 (from #tr)))
531         by (apply H1; exact Htrans).
532       assert (Hb2:  if_software (labelled #tr)
533                                 (software_req hse_2 (from #tr)))
534         by (apply H2; exact Htrans).
535       induction tr as [tr _H3]; induction tr as [[h l] h'].
536       induction l; cbn in *; auto.
537   Qed.
```

A.3 Case Study: Code Injection Policies

A.3.1 The Software Stack

We consider the execution of a software stack made of a firmware component, an operating system and an infinite number of applications, identified by a natural number.

```
262   Inductive software_stack
263   : Type :=
```

```
| BIOS
  : software_stack
| OS
  : software_stack
| App (n:  nat)
  : software_stack.
```

Because our hardware model is left as a parameter of this Coq development, so are the hardware-software mapping and transition-software mapping that we will use to reason about the software stack execution.

```
Variables (ss_context:  H -> software_stack)
          (ss_fetched:  H -> label -> (software_stack -> Prop)).
```

A.3.2 Code Injection

Implementing the code injection definition (Definition 4.11) in Gallina is straightforward.

```
Definition code_injection
           {m:    model}
           (tr:   transition m)
           (x y:  software_stack)
  : Prop :=
  ss_fetched (from #tr) (labelled #tr) y
  /\ ss_context (from #tr) = y.
```

A.3.3 Code Injection Policies

The code injection policy is defined against a relation between software components to tell whether one software component is authorized to perform a code injection against another. In our case, we implement this relation as an inductive predicate.

```
Inductive stack_ge
  : software_stack -> software_stack -> Prop :=
| stack_ge_refl (x:  software_stack)
  : stack_ge x x
| bios_bottom (x:  software_stack)
  : stack_ge BIOS x
| os_apps (n:  nat)
  : stack_ge OS (App n).
```

The proof that the relation `stack_ge` is antisymmetric relies on a case analysis.

```
586 Lemma stack_ge_anti_sym
587       (x y:  software_stack)
588   : stack_ge x y -> stack_ge y x -> x = y.
589 Proof.
590   intros H1 H2.
591   inversion H1; subst; inversion H2; subst; try reflexivity.
592 Qed.
```

Using these components, the definition of the code injection policy is simple.

```
346 Definition code_injection_policy
347            {m:   model}
348            (tr:  transition m)
349   : Prop :=
350   forall (x y:  software_stack),
351     code_injection tr x y
352     -> stack_ge x y.
```

As explained in Chapter 4, the enforcement of such security policy is done thanks to several HSE mechanisms. A first HSE mechanism will be devoted to enforce that the BIOS remains isolated from the rest of the software stack, that is it will enforce the following policy:

```
539 Definition bios_code_injection_policy
540            {m:         model}
541            (tr:  transition m)
542   : Prop :=
543   forall (x:  software_stack),
544     code_injection tr x BIOS
545     -> x = BIOS.
```

A second HSE mechanism will be devoted to enforce that the applications cannot perform illegitimate code injection against the rest of the software stack, that is:

```
547 Definition os_code_injection_policy
548            {m:         model}
549            (tr:  transition m)
550   : Prop :=
551   forall (x:  software_stack)
552          (n:  nat),
553     code_injection tr (App n) x
554     -> x = App n.
```

Our theory of HSE mechanisms allows us to prove that the concurrent implementation of two HSE mechanisms —correct with respect to their respective security policy— is a sufficient

condition for the enforcement "global" code injection policies, by demonstrating that the composition of these HSE mechanisms is correct with respect to the code injection policy.

```
Theorem constrain_everyone
        {m:         model}
        (hse_bios:  HSE m)
        (hse_os:    HSE m)
        (Hcomp:     compatible_hse hse_bios hse_os)
  : correct_hse hse_bios
                (safety_property bios_code_injection_policy)
    -> correct_hse hse_os
                   (safety_property os_code_injection_policy)
    -> correct_hse (HSE_cap hse_bios hse_os Hcomp)
                   (safety_property code_injection_policy).
Proof.
  intros Hbios Hos rho Hct tr Htrans.
  apply compliant_trace_intersec_intersec_compliant_trace in Hct.
  inversion Hct as [Hrb Hro].
  intros x y Htamper.
  unfold correct_hse, safety_property in *.
  unfold bios_code_injection_policy, os_code_injection_policy in *.
  induction x.
  + constructor.
  + induction y.
    ++ apply (Hbios rho Hrb tr Htrans) in Htamper.
       discriminate.
    ++ constructor.
    ++ constructor.
  + apply (Hos rho Hro tr Htrans) in Htamper.
    rewrite Htamper.
    constructor.
```

B

Publications

B.1 Peer-reviewed Conferences

– **SpecCert: Specifying and Verifying Hardware-based Security Enforcement Mechanisms.**
Thomas Letan, Pierre Chifflier, Guillaume Hiet, Pierre Néron, Benjamin Morin, *21st International Symposium on Formal Methods (FM 2016)*.

– **Modular Verification of Programs with Effects and Effect Handlers in Coq.**
Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, Guillaume Hiet, *22st International Symposium on Formal Methods (FM 2018)*.

B.2 Free Software

– **SpecCert**, a Coq framework for specifying and verifying Hardware-based Security Enforcement, distributed under the terms of the CeCILL-B — https://github.com/lthms/speccert

– **FreeSpec**, a Coq framework for modularly verifying programs with effects and effect handlers, distributed under the terms of the GPL-3 — https://github.com/ANSSI-FR/FreeSpec

B.3 Seminar

– **FreeSpec : Modular Verification of Systems using Effects and Effect Handlers in Coq**, *Analyse et conception de systèmes, IRIF* – https://www.irif.fr/seminaires/acs/index

Acronyms

ACPI Advanced Configuration and Power Interface

BIOS Basic Input/Output System

CPU Central Processing Unit

DMA Direct Memory Access

DRAM Dynamic Random Access Memory

HSE Hardware-based Security Enforcement

IDT Interrupt Descriptor Table

LTS Labelled Transition System

MMU Memory Management Unit

PCH Platform Controller Hub

PMIO Port-Mapped I/O

SMI System Management Interrupt

SMM System Management Mode

SMRR System Manangement Range Registers

TCB Trusted Computing Base

UEFI Unified Extensible Firmware Interface

XOM eXecute Only Memory

List of Figures

List of Tables

Bibliography

[1] Xeno Kovah, Corey Kallenberg, John Butterworth, and Sam Cornwell. SENTER Sandman: Using Intel TXT to Attack Bioses. *Hack in the Box*, 2015.

[2] Intel. *Intel Trusted Execution Technology (Intel TXT)*. 07 2015.

[3] Jeannette M Wing. A Call to Action: Look beyond the Horizon. *IEEE Security & Privacy*, (6):62–67, 2003.

[4] Loic Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. Getting into the SMRAM: SMM Reloaded. CanSecWest.

[5] Rafal Wojtczuk and Joanna Rutkowska. Attacking SMM Memory via Intel CPU Cache Poisoning.

[6] Christopher Domas. The Memory Sinkhole. In *BlackHat USA*, july 2015.

[7] Corey Kallenberg and Rafal Wojtczuk. Speed Racer: Exploiting an Intel Flash Protection Race Condition. January 2015.

[8] Ulrich Stern and David L Dill. Automatic Verification of the SCI Cache Coherence Protocol. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 21–34. Springer, 1995.

[9] Muralidaran Vijayaraghavan, Adam Chlipala, Nirav Dave, et al. Modular Deductive Verification of Multiprocessor Hardware Designs. In *International Conference on Computer Aided Verification*, pages 109–127. Springer, 2015.

[10] Joonwon Choi, Muralidaran Vijayaraghavan, Benjamin Sherman, Adam Chlipala, et al. Kami: A Platform for High-Level Parametric Hardware Specification and Its Modular Verification. *Proceedings of the ACM on Programming Languages*, 1(ICFP):24, 2017.

[11] Narjes Jomaa, David Nowak, Gilles Grimaud, and Samuel Hym. Formal Proof of Dynamic Memory Isolation Based on MMU. In *Theoretical Aspects of Software Engineering (TASE), 2016 10th International Symposium on*, pages 73–80. IEEE, 2016.

[12] The Pip Development Team. The pip protokernel. http://pip.univ-lille1.fr/.

[13] Deepak Garg, Jason Franklin, Dilsun Kaynar, and Anupam Datta. Compositional System Security with Interface-Confined Adversaries. *Electronic Notes in Theoretical Computer Science*, 265:49–71, 2010.

[14] Thomas Heyman, Riccardo Scandariato, and Wouter Joosen. Reusable Formal Models for Secure Software Architectures. In *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*, pages 41–50. IEEE, 2012.

[15] Thomas Letan, Pierre Chifflier, Guillaume Hiet, Pierre Néron, and Benjamin Morin. SpecCert: Specifying and Verifying Hardware-based Security Enforcement. In *21st International Symposium on Formal Methods (FM 2016)*. Springer, 2016.

[16] Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, and Guillaume Hiet. Modular Verification of Programs with Effects and Effect Handlers. In *28st International Symposium on Formal Methods (FM 2018)*. Springer, 2018.

[17] Leslie Lamport. Proving the Correctness of Multiprocess Programs. *IEEE transactions on software engineering*, (2):125–143, 1977.

[18] Leslie Lamport. Logical Foundation. *Distributed systems-methods and tools for specification*, 190:119–130, 1985.

[19] Bowen Alpern and Fred B Schneider. Defining Liveness. Technical report, Cornell University, 1985.

[20] Michael R Clarkson and Fred B Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[21] Thomas Letan. FreeSpec: a Compositional Reasoning Framework for the Coq Theorem Prover. https://github.com/lthms/speccert.

[22] Inria. The Coq Proof Assistant. https://coq.inria.fr/.

[23] Simon Peyton Jones. Tackling the Awkward Squad: monadic I/O, concurrency, exception and foreign-language calls in Haskell. *Engineering theories of software construction*, pages 47–96, 2005.

[24] Andrej Bauer and Matija Pretnar. Programming with Algebraic Effects and Handlers. *Journal of Logical and Algebraic Methods in Programming*, 84(1):108–123, 2015.

[25] Kaiyuan Yang, Matthew Hicks, Qing Dong, Todd Austin, and Dennis Sylvester. A2: Analog Malicious Hardware. In *Security and Privacy (SP), 2016 IEEE Symposium on*, pages 18–37. IEEE, 2016.

[26] Shawn Embleton, Sherri Sparks, and Cliff C Zou. SMM rootkit: a new breed of OS independent malware. *Security and Communication Networks*, 6(12):1590–1605, 2013.

[27] Yuriy Bulygin, J Loucaides, Andrew Furtak, O Bazhaniuk, and A Matrosov. Summary of Attacks Against BIOS and Secure Boot. *Proceedings of the DefCon*, 2014.

[28] MITRE. CVE-2018-8897. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-8897.

[29] Intel. *Intel 64 and IA32 Architectures Software Developer Manual*, chapter Introduction to Virtual Machine Extensions. October 2014.

[30] Intel. *Intel 64 and IA32 Architectures Software Developer Manual*, chapter Introduction to Intel Software Guard Extenions. October 2014.

[31] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptology ePrint Archive*, 2016:86, 2016.

[32] Rafal Wojtczuk and Joanna Rutkowska. Attacking Intel TXT via SINIT Code Execution Hijacking. November 2011.

[33] Fernand Lone Sang, Eric Lacombe, Vincent Nicomette, and Yves Deswarte. Exploiting an I/OMMU vulnerability. In *Malicious and Unwanted Software (MALWARE), 2010 5th International Conference on*, pages 7–14. IEEE, 2010.

[34] Intel. *Intel 64 and IA32 Architectures Software Developer Manual*. October 2014.

[35] Intel. *Intel 5100 Memory Controller Hub Chipset*.

[36] Intel. *Intel 7 Series / C216 Chipset Family Platform Controller Hub (PCH)*.

[37] Thomas Letan. SpecCert: a framework for the Coq Theorem Prover. https://github.com/lthms/speccert.

[38] Michael E Thomadakis. The Architecture of the Nehalem Processor and Nehalem-EP SMP Platforms. *Resource*, 3(2), 2011.

[39] J Turley. White Paper Introduction to Intel® Architecture, 2014.

[40] Debbie Marr, Frank Binns, D Hill, Glenn Hinton, D Koufaty, et al. Hyper-Threading Technology in the Netburst® Microarchitecture. *14th Hot Chips*, 2002.

[41] Agner Fog. The Microarchitecture of Intel, AMD and VIA CPUs: An Optimization Guide for Assembly Programmers and Compiler Makers. *Copenhagen University College of Engineering*, 2012.

[42] Milena Milenkovic, Aleksandar Milenkovic, and Jeffrey Kulick. Demystifying Intel Branch Predictors. In *Workshop on Duplicating, Deconstructing and Debunking*, 2002.

[43] Simon P Johnson, Uday R Savagaonkar, Vincent R Scarlata, Francis X McKeen, and Carlos V Rozas. Technique for Supporting Multiple Secure Enclaves, 2015. US Patent 8,972,746.

[44] Jerome H Saltzer and Michael D Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[45] Darren Abramson, Jeff Jackson, Sridhar Muthrasanallur, Gil Neiger, Greg Regnier, Rajesh Sankaran, Ioannis Schoinas, Rich Uhlig, Balaji Vembu, and John Wiegert. Intel Virtualization Technology for Directed I/O. *Intel technology journal*, 10(3), 2006.

[46] Marion Daubignard and Yves-Alexis Perez. ProTIP: You Should Know What to Expect From Your Peripherals. *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, June 2017.

[47] Karsten Nohl and Jakob Lell. BadUSB – On Accessories That Turn Evil. *Black Hat USA*, 2014.

[48] Trammell Hudson and Larry Rudolph. Thunderstrike: EFI Firmware Bootkits for Apple MacBooks. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 15. ACM, 2015.

[49] Pierre Chifflier. UEFI et bootkits PCI: le danger vient d'en bas. In *Actes du 11ème symposium sur la sécurité des technologies de l'information et des communications (SSTIC)*, pages 159–190, 2013.

[50] Darmawan Salihun. *BIOS Disassembly Ninjutsu Uncovered (Uncovered series)*. A-List Publishing, 2006.

[51] Vincent Zimmer. Platform Trust Beyond BIOS Using the Unified Extensible Firmware Interface. In *Security and Management*, pages 400–405, 2007.

[52] UEFI Forum. *Unified Extensible Firmware Interface (UEFI) Specification, Version 2.7 Errata A*. August 2017.

[53] Matt Fleming. [PATCH] x86 EFI boot stub, October 2011. https://lkml.org/lkml/2011/10/17/81.

[54] Joanna Rutkowska. Intel x86 Considered Harmful. *the Invisible Things Lab*, 2015.

[55] Lee Rosenbaum and Vincent Zimmer. A Tour Beyond BIOS into UEFI Secure Boot. 2012.

[56] Jonathan Corbet. Protecting systems with the TPM, February 2016. https://lwn.net/Articles/674751/.

[57] HP Inc. HP Sure Start: Automatic Firmware Intrusion Detection and Repair System, 2016.

[58] Andrew Regenscheid. Platform Firmware Resiliency Guidelines. *NIST Special Publication*, 800:193, 2018.

[59] UEFI Forum. *Advanced Configuration and Power Interface (ACPI) Specification, Version 6.2 Errata A*. September 2017.

[60] Loïc Duflot, Olivier Grumelard, Olivier Levillain, and Benjamin Morin. ACPI and SMI Handlers: Some Limits to Trusted Computing. *Journal in computer virology*, 6(4):353–374, 2010.

[61] Vincent Zimmer, Michael Rothman, and Suresh Marisetty. *Beyond BIOS: Developing with the Unified Extensible Firmware Interface*. Walter de Gruyter GmbH & Co KG, 2017.

[62] Jiewen Yao, Vincent J Zimmer, and Qin Long. System Management Mode Isolation in Firmware, May 7 2009. US Patent App. 12/317,446.

[63] Aarti Gupta. Formal Hardware Verification Methods: A Survey. In *Computer-Aided Verification*, pages 5–92. Springer, 1992.

[64] Rebekah Leslie-Hurd, Dror Caspi, and Matthew Fernandez. Verifying Linearizability of Intel® Software Guard Extensions. In *International Conference on Computer Aided Verification*, pages 144–160. Springer, 2015.

[65] David Lie, John Mitchell, Chandramohan A Thekkath, and Mark Horowitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *Security and Privacy, 2003. Proceedings. 2003 Symposium on*, pages 166–177. IEEE, 2003.

[66] Claire Loiseaux, Susanne Graf, Joseph Sifakis, Ahmed Bouajjani, Saddek Bensalem, and David Probst. Property Preserving Abstractions for the Verification of Concurrent Systems. *Formal methods in system design*, 6(1):11–44, 1995.

[67] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV2: An Opensource Tool for Symbolic Model Checking. In *International Conference on Computer Aided Verification*, pages 359–364. Springer, 2002.

[68] Daniel Jackson. *Software Abstractions: Logic, Language and Analysis*. MIT press, 2012.

[69] University of Utah School of Computing. Murphi Model Checker. https://formalverification.cs.utah.edu/Murphi/.

[70] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Formally Verifying Isolation and Availability in an Idealized Model of Virtualization. In *International Symposium on Formal Methods*, pages 231–245. Springer, 2011.

[71] Bowen Alpern and Fred B Schneider. Recognizing Safety and Liveness. *Distributed computing*, 2(3):117–126, 1987.

[72] Fred B. Schneider. Enforceable Security Policies. *ACM Transactions on Information and System Security (TISSEC)*, 3(1):30–50, 2000.

[73] David Basin, Vincent Jugé, Felix Klaedtke, and Eugen Zălinescu. Enforceable Security Policies Revisited. *ACM Transactions on Information and System Security (TISSEC)*, 16(1):3, 2013.

[74] Joseph A Goguen and José Meseguer. Security Policies and Security Models. In *Security and Privacy, 1982 IEEE Symposium on*, pages 11–11. IEEE, 1982.

[75] Edmund M Clarke, Thomas A Henzinger, and Helmut Veith. Introduction to Model Checking. In *Handbook of Model Checking*, pages 1–26. Springer, 2018.

[76] Carla P Gomes, Henry Kautz, Ashish Sabharwal, and Bart Selman. Satisfiability Solvers. *Foundations of Artificial Intelligence*, 3:89–134, 2008.

[77] Edmund M Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model Checking and the State Explosion Problem. In *Tools for Practical Software Verification*, pages 1–30. Springer, 2012.

[78] Raymond R Smullyan. *First-Order Logic*, volume 43. Springer Science & Business Media, 2012.

[79] Sharad Malik and Lintao Zhang. Boolean Satisfiability From Theoretical Hardness to Practical Success. *Communications of the ACM*, 52(8):76–82, 2009.

[80] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[81] Clark Barrett, Morgan Deters, Leonardo De Moura, Albert Oliveras, and Aaron Stump. 6 Years of SMT-COMP. *Journal of Automated Reasoning*, 50(3):243–277, 2013.

[82] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[83] Alexander Chagrov. Modal Logic. 1997.

[84] A Prasad Sistla and Edmund M Clarke. The Complexity of Propositional Linear Temporal Logics. *Journal of the ACM (JACM)*, 32(3):733–749, 1985.

[85] Edmund M Clarke and E Allen Emerson. Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[86] Gerard J. Holzmann. The Model Checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[87] Leslie Lamport. *Specifying Systems: the TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[88] Kenneth L McMillan. Symbolic Model Checking. In *Verification of Digital and Hybrid Systems*, pages 117–137. Springer, 2000.

[89] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, Yunshan Zhu, et al. Bounded Model Checking. *Advances in computers*, 58(11):117–148, 2003.

[90] Daniel Leivant. Higher Order Logic, 1994.

[91] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[92] Jean-Raymond Abrial and Jean-Raymond Abrial. *The B-book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[93] Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris Van Doorn, and Jakob von Raumer. The Lean Theorem Prover (System Description). In *International Conference on Automated Deduction*, pages 378–388. Springer, 2015.

[94] John Harrison. Formal Verification of IA-64 Division Algorithms. In *International Conference on Theorem Proving in Higher Order Logics*, pages 233–251. Springer, 2000.

[95] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik Reeber, et al. Replacing Testing with Formal Verification in Intel^{\scriptsize\circledR} CoreTM i7 Processor Execution Engine Validation. In *International Conference on Computer Aided Verification*, pages 414–429. Springer, 2009.

[96] Nachiketh Potlapally. Hardware Security in Practice: Challenges and Opportunities. In *Hardware-Oriented Security and Trust (HOST), 2011 IEEE International Symposium on*, pages 93–98. IEEE, 2011.

[97] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, Faster, Stronger SFI for the x86. In *ACM SIGPLAN Notices*, volume 47, pages 395–404. ACM, 2012.

[98] Shilpi Goel, Warren A Hunt, Matt Kaufmann, and Soumava Ghosh. Simulation And Formal Verification Of x86 Machine-Code Programs That Make System Calls. In *Formal Methods in Computer-Aided Design (FMCAD), 2014*, pages 91–98. IEEE, 2014.

[99] Xavier Leroy et al. The CompCert Verified Compiler. *Documentation and user's manual. INRIA Paris-Rocquencourt*, 2012.

[100] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan (Newman) Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *OSDI*, volume 16, pages 653–669, 2016.

[101] Hao Chen, Xiongnan Wu, Zhong Shao, Joshua Lockerman, and Ronghui Gu. Toward Compositional Verification of Interruptible OS Kernels and Device Drivers. *Journal of Automated Reasoning*, 61(1-4):141–189, 2018.

[102] Data61/CSIRO. The seL4 Microkernel. https://sel4.systems/.

[103] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220. ACM, 2009.

[104] Gilles Barthe, Gustavo Betarte, Juan Diego Campo, and Carlos Luna. Cache-Leakage Resilient OS Isolation in an Idealized Model of Virtualization. In *Computer Security Foundations Symposium (CSF), 2012 IEEE 25th*, pages 186–197. IEEE, 2012.

[105] Gilles Barthe, Gustavo Betarte, Juan Campo, Carlos Luna, and David Pichardie. System-level Non-Interference for Constant-Time Cryptography. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1267–1279. ACM, 2014.

[106] Anthony Fox and Magnus O Myreen. A Trustworthy Monadic Formalization of the ARMv7 Instruction Set Architecture. In *International Conference on Interactive Theorem Proving*, pages 243–258. Springer, 2010.

[107] Alastair Reid. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*, pages 161–168. FMCAD Inc, 2016.

[108] Alastair Reid, Rick Chen, Anastasios Deligiannis, David Gilday, David Hoyes, Will Keen, Ashan Pathirane, Owen Shepherd, Peter Vrabel, and Ali Zaidi. End-to-end Verification of Processors with ISA-Formal. In *International Conference on Computer Aided Verification*, pages 42–58. Springer, 2016.

[109] ARM Ltd. A64 ISA XML for Armv8.4, 2018.

[110] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. *ACM SIGPLAN Notices*, 35(11):168–177, 2000.

[111] Amir Pnueli. In Transition from Global to Modular Temporal Reasoning about Programs. In *Logics and models of concurrent systems*, pages 123–144. Springer, 1985.

[112] Cliff B. Jones. Tentative Steps Toward a Development Method for Interfering Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):596–619, 1983.

[113] Luca De Alfaro and Thomas A Henzinger. Interface Automata. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 109–120. ACM, 2001.

[114] Robin Milner. A Calculus of Communicating Systems. *LNCS*, 92, 1980.

[115] Charles Antony Richard Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[116] Rishiyur Nikhil. Bluespec System Verilog: efficient, correct RTL from high level specifications. In *Formal Methods and Models for Co-Design, 2004. MEMOCODE'04. Proceedings. Second ACM and IEEE International Conference on*, pages 69–70. IEEE, 2004.

[117] Leslie Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Progranm. *IEEE transactions on computers*, (9):690–691, 1979.

[118] Jeanine Souquieres. Verifying the Compatibility of Component Interfaces using the B Formal Method. *Software Engineering Research and Practice*, pages 850–856, 2005.

[119] Samir Chouali, Maritta Heisel, and Jeanine Souquières. Proving Component Interoperability with B Refinement. *Electronic Notes in Theoretical Computer Science*, 160:157–172, 2006.

[120] Arnaud Lanoix and Jeanine Souquières. Component-based Development using the B method, July 2006. Research report.

[121] Guillaume Claret. Coq.io. https://coq.io.

[122] Philip Wadler. Comprehending Monads. In *Proceedings of the 1990 ACM conference on LISP and functional programming*, pages 61–78. ACM, 1990.

[123] Martin Hyland, Gordon Plotkin, and John Power. Combining Effects: Sum and Tensor. *Theoretical Computer Science*, 357(1-3):70–99, 2006.

[124] Sheng Liang, Paul Hudak, and Mark Jones. Monad Transformers and Modular Interpreters. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 333–343. ACM, 1995.

[125] Edwin Brady. Programming and Reasoning with Algebraic Effects and Dependent Types. In *ACM SIGPLAN Notices*, volume 48, pages 133–144. ACM, 2013.

[126] Oleg Kiselyov, Amr Sabry, and Cameron Swords. Extensible Effects: An Alternative To Monad Transformers. In *ACM SIGPLAN Notices*, volume 48, pages 59–70. ACM, 2013.

[127] Joanna Rutkowska and Rafal Wojtczuk. Preventing and Detecting Xen Hypervisor Subversions. *Blackhat Briefings USA*, 2008.

[128] Intel. *Desktop 4th Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family*.

[129] Peter Müller, Arnd Poetzsch-Heffter, and Gary T Leavens. Modular Invariants for Layered Object Structures. *Science of Computer Programming*, 62(3):253–286, 2006.

[130] Heinrich Apfelmus. The `operational` package, 2010. https://hackage.haskell.org/package/operational.

[131] CAR Hoareetal. Tackling the Awkward Squad: monadic input/output, concurrency, exceptions, and foreign-language calls in Haskell. *Engineering theories of software construction*, 2001.

[132] Jiri Gaisler. LEON SPARC Processor: The Past, Present and Future. *RAMP Winter Retreat, Berkeley*, 2007.

Titre : Spécifier et vérifier des stratégies d'application de politiques de sécurité s'appuyant sur des mécanismes matériels

Mots clés : Sécurité ▪ Vérification matérielle ▪ Méthodes formelles ▪ Coq

Résumé : Dans ces travaux de thèse, nous nous intéressons à une classe de stratégies d'application de politiques de sécurité que nous appelons HSE, pour *Hardware-based Security Enforcement*. Dans ce contexte, un ou plusieurs composants logiciels de confiance contraignent l'exécution du reste de la pile logicielle avec le concours de la plate-forme matérielle sous-jacente afin d'assurer le respect d'une politique de sécurité donnée.
Pour qu'un mécanisme HSE contraigne effectivement l'exécution de logiciels arbitraires, il est nécessaire que la plate-forme matérielle et les composants logiciels de confiance l'implémentent correctement. Ces dernières années, plusieurs vulnérabilités ont mis à défaut des implémentations de mécanismes HSE. Nous concentrons ici nos efforts sur celles qui sont le résultat d'erreurs dans les spécifications matérielles et non dans une implémentation donnée.
Plus précisément, nous nous intéressons aux cas particulier de l'usage légitime, par un attaquant, d'une fonctionnalité d'un composant matériel pour contourner les protections offertes par un second.
Notre but est d'explorer des approches basées sur l'usage de méthodes formelles pour spécifier et vérifier des mécanismes HSE. La spécification de mécanismes HSE peut servir de point de départ pour la vérification des spécifications matérielles concernées, dans l'espoir de prévenir des attaques profitant de la composition d'un grand nombre de composants matériels. Elles peuvent ensuite être fournies aux développeurs logiciels, sous la forme d'une liste de prérequis que leurs produits doivent respecter s'ils désirent l'application d'une politique de sécurité clairement identifiée.

Title : Specifying and Verifying Hardware-based Security Enforcement Mechanisms

Keywords: Security ▪ Hardware Verification ▪ Formal Methods ▪ Coq

Abstract: In this thesis, we consider a class of security enforcement mechanisms we called Hardware-based Security Enforcement (HSE). In such mechanisms, some trusted software components rely on the underlying hardware architecture to constrain the execution of untrusted software components with respect to targeted security policies. For instance, an operating system which configures page tables to isolate userland applications implements a HSE mechanism.
For a HSE mechanism to correctly enforce a targeted security policy, it requires both hardware and trusted software components to play their parts. During the past decades, several vulnerability disclosures have defeated HSE mechanisms. We focus on the vulnerabilities that are the result of errors at the specification level, rather than implementation errors. In some critical vulnerabilities, the attacker makes a legitimate use of one hardware component to circumvent the HSE mechanism provided by another one. For instance, cache poisoning attacks leverage inconsistencies between cache and DRAM's access control mechanisms. We call this class of attacks, where an attacker leverages inconsistencies in hardware specifications, compositional attacks.
Our goal is to explore approaches to specify and verify HSE mechanisms using formal methods that would benefit both hardware designers and software developers. Firstly, a formal specification of HSE mechanisms can be leveraged as a foundation for a systematic approach to verify hardware specifications, in the hope of uncovering potential compositional attacks ahead of time. Secondly, it provides unambiguous specifications to software developers, in the form of a list of requirements.